



SISTEMAS OPERATIVOS DIVTIC DO4

ALGORITMO DE DEKKER Y PETERSON

PROFESORA

VIOLETA DEL ROCIO BECERRA VELAZQUEZ

EQUIPO 2

VAZQUEZ DELGADO KEVIN
GUTIERREZ VAZQUEZ AXEL
VERDUZCO ROSALES LUIS ENRIQUE
QUINTERO ARREOLA LAURA VANESSA
MORALES GUTIERREZ ALAN OSWALDO

12/10/2025



Índice

1. Fundamentos de la Sincronización de Procesos: El Problema de la Sección Crítica.....	2
2. El Algoritmo de Dekker: La Primera Solución de Software Probablemente Correcta.....	4
3. El Algoritmo de Peterson: Una Simplificación Elegante	8
4. Análisis Comparativo y Limitaciones Fundamentales.....	14
5. Soluciones Modernas al Problema de la Sección Crítica.....	16
6. Conclusión: El Legado de Dekker y Peterson en la Computación Concurrente.....	18
7. Bibliografía.....	19

1. Fundamentos de la Sincronización de Procesos: El Problema de la Sección Crítica

- Introducción a la Programación Concurrente

La programación concurrente es la rama de la informática que se ocupa de las técnicas de programación utilizadas para expresar el paralelismo entre tareas y para resolver los problemas de comunicación y sincronización entre procesos que se ejecutan simultáneamente. En un sistema concurrente, múltiples procesos o hilos de ejecución pueden avanzar en sus tareas de forma superpuesta en el tiempo, ya sea en un sistema monoprocesador mediante la intercalación de su ejecución (multitarea) o en un sistema multiprocesador mediante la ejecución paralela real. El desafío fundamental de este paradigma radica en el no determinismo del orden de ejecución de las instrucciones. Sin mecanismos de control adecuados, este orden impredecible puede llevar a resultados incorrectos y a fallos de software difíciles de reproducir y depurar.

En el núcleo de la programación concurrente se encuentra la gestión de recursos. Los procesos compiten por el uso de recursos limitados, que pueden clasificarse en dos categorías principales: compartibles y no compartibles. Los recursos compartibles, como los archivos de solo lectura o las áreas de memoria no modificables, pueden ser utilizados por varios procesos de forma simultánea sin riesgo de conflicto. Por el contrario, los recursos no compartibles, como una impresora, una unidad de cinta magnética o una variable en memoria que puede ser modificada, deben ser accedidos de manera exclusiva. El uso concurrente de estos recursos no compartibles puede provocar que la acción de un proceso interfiera con la de otro, dando lugar a un estado del sistema inconsistente o corrupto.

- El Origen de las Condiciones de Carrera (Race Conditions)

El acceso concurrente no gestionado a recursos compartidos, particularmente a datos en memoria, es la fuente de una clase de errores conocidos como "condiciones de carrera" (*race conditions*). Una condición de carrera ocurre cuando el resultado de un cómputo depende del orden o temporización de eventos incontrolables, como la planificación de procesos por parte del sistema operativo. Estos errores surgen porque las operaciones que parecen ser una única instrucción en un lenguaje de alto nivel, en realidad se descomponen en una secuencia de múltiples instrucciones a nivel de lenguaje de máquina.

Un ejemplo clásico que ilustra este problema es la operación de incremento de un contador compartido, como `count++`. A nivel de máquina, esta simple operación se traduce típicamente en tres instrucciones distintas:

1. **LOAD:** Cargar el valor actual de la variable `count` desde la memoria a un registro de la CPU.
2. **INCREMENT:** Incrementar el valor contenido en el registro.
3. **STORE:** Almacenar el nuevo valor del registro de vuelta en la ubicación de memoria de `count`.

Si dos procesos, P0 y P1, intentan ejecutar `count++` de forma concurrente, puede ocurrir una intercalación perjudicial. Supongamos que `count` es inicialmente 5. P0 ejecuta la instrucción **LOAD** y lee el valor 5 en su registro. Antes de que P0 pueda continuar, el planificador del

sistema operativo le quita la CPU y se la asigna a P1 (una interrupción). P1 ejecuta las tres instrucciones completas: carga 5, lo incrementa a 6 y lo almacena de nuevo en memoria. Ahora, count es 6. Cuando P0 recupera la CPU, reanuda su ejecución desde donde la dejó. Ya había cargado el valor 5 en su registro, por lo que procede a incrementarlo a 6 y lo almacena en memoria. El resultado final es que count vale 6, cuando el resultado correcto debería ser 7, ya que dos operaciones de incremento se realizaron. La actualización de P1 se ha perdido. Este tipo de interferencia es de vital importancia, ya que puede provocar fallos graves y sutiles en el sistema.

- Formalización del Problema: La Sección Crítica

Para abordar sistemáticamente las condiciones de carrera, es necesario identificar las partes del código de un programa que son vulnerables. Una **sección crítica (SC)** se define como una secuencia de instrucciones dentro de un proceso durante la cual se accede a uno o más recursos compartidos (como variables o estructuras de datos) que no deben ser accedidos concurrentemente por otros procesos. La ejecución del código dentro de una sección crítica debe ser mutuamente excluyente en el tiempo.

El "problema de la sección crítica" consiste en diseñar un protocolo que los procesos puedan utilizar para cooperar. Cada proceso debe solicitar permiso para entrar en su sección crítica. El código que implementa esta solicitud se denomina **sección de entrada**. La sección crítica puede ir seguida de una **sección de salida**, donde se realizan las acciones necesarias para permitir que otros procesos entren en sus propias secciones críticas. El resto del código del proceso se denomina **sección restante**. El objetivo es desarrollar un protocolo de entrada y salida que garantice la integridad de los datos compartidos.

- Requisitos Canónicos para una Solución Válida

Cualquier solución robusta al problema de la sección crítica debe satisfacer tres condiciones esenciales, que se han convertido en el estándar canónico para evaluar la corrección de los algoritmos de sincronización.

1. **Exclusión Mutua (Mutual Exclusion):** Esta es la propiedad de seguridad fundamental. Establece que si un proceso se está ejecutando en su sección crítica, ningún otro proceso puede estar ejecutándose simultáneamente en su respectiva sección crítica. Garantiza que el acceso al recurso compartido sea exclusivo, evitando así las condiciones de carrera.
2. **Progreso (Progress):** Esta es una propiedad de "liveness" o vivacidad para el sistema en su conjunto. Si ningún proceso está ejecutando en su sección crítica y existe un conjunto de procesos que desean entrar, la decisión sobre cuál de estos procesos será el siguiente en entrar no puede posponerse indefinidamente. Además, esta decisión solo puede involucrar a los procesos que están esperando para entrar, no a aquellos que se encuentran en su

sección restante. Esta condición asegura que el sistema no se detenga si hay trabajo por hacer.

3. **Espera Limitada (Bounded Waiting):** Esta es una propiedad de "liveness" y equidad para cada proceso individual. Debe existir un límite en el número de veces que a otros procesos se les permite entrar en sus secciones críticas después de que un proceso haya hecho una solicitud para entrar en la suya y antes de que dicha solicitud sea concedida. Esta condición previene la inanición (*starvation*), una situación en la que a un proceso se le niega continuamente el acceso a un recurso a pesar de que este se otorga a otros procesos.

Es importante discernir la sutil pero crucial diferencia entre progreso y espera limitada. Una solución puede satisfacer la condición de progreso pero no la de espera limitada. Por ejemplo, si un algoritmo siempre elige entre dos procesos en espera, P0 y P1, al proceso P0, el sistema en su conjunto progresa (P0 entra repetidamente en su sección crítica), pero P1 sufre de inanición, ya que su espera no está limitada. Por lo tanto, la espera limitada es una garantía de equidad mucho más fuerte que el progreso. Previene el bloqueo indefinido de un proceso individual, mientras que el progreso solo previene el bloqueo de todo el grupo de procesos en espera. La evolución de los algoritmos de sincronización, como se verá en el caso de Dekker, a menudo implica refinar la lógica para pasar de simplemente garantizar el progreso a asegurar también una espera limitada.

Finalmente, todas las soluciones de software, como las de Dekker y Peterson, se basan en una suposición implícita pero fundamental: que las operaciones de lectura y escritura de una palabra de memoria son atómicas. Es decir, una instrucción como `flag[i] = true` se ejecuta de manera indivisible. Si una escritura no fuera atómica, otro proceso podría leer un valor intermedio e inconsistente, invalidando por completo la lógica del algoritmo. Esta dependencia del hardware subyacente prefigura la eventual transición hacia soluciones que utilizan instrucciones atómicas más potentes proporcionadas por el propio hardware.

2. El Algoritmo de Dekker: La Primera Solución de Software Probablemente Correcta

Contexto Histórico: La Búsqueda de una Solución Puramente de Software

En los inicios de la computación multitarea, surgió un problema importante, el cuál era evitar que varios procesos chocaran entre sí al intentar usar el mismo recurso compartido, de esta forma, si dos procesos modificaban un dato a la vez, se podían generar "condiciones de carrera", corrompiendo la información y causando un caos impredecible en el sistema. La solución al problema era garantizar la exclusión mutua, es decir, que solo un proceso pudiera acceder a esa "sección crítica" (el fragmento de código que maneja el recurso compartido) en un momento dado.

Las primeras ideas dependían de hardware específico, lo que limitaba su uso a ciertas máquinas y por eso era necesaria una solución puramente de software, entonces, es aquí donde entra en escena el matemático holandés Theodorus Jozef Dekker, quien en 1965 desarrolló uno de los primeros algoritmos para resolver este problema, El Algoritmo de Dekker.

La Evolución del Algoritmo: Un Estudio de Caso en Refinamiento Iterativo (Primeras 4 versiones)

Versión 1. Alternancia Estricta

La primera versión utiliza una variable de turno (turn) para alternar estrictamente el acceso a la sección crítica entre dos procesos (P0 y P1).

Características:

- Una variable compartida turn se inicializa en 0 o 1.
- El proceso P0 solo puede entrar a su sección crítica si turn es 0.
- El proceso P1 solo puede entrar si turn es 1.
- Después de salir de la sección crítica, el proceso que la ocupaba cede el turno al otro (si P0 sale, establece turn = 1; si P1 sale, establece turn = 0).

Problema con la versión 1:

El principal problema de esta versión es que fuerza a los procesos a alternar, incluso si uno de ellos no necesita entrar a la sección crítica. Si, por ejemplo, es el turno de P1 (turn = 1), pero P1 no tiene interés en entrar a su sección crítica, P0 no podrá acceder, aunque el recurso esté libre.

Versión 2. Banderas de Intención

Esta versión intenta solucionar el problema de la alternancia estricta utilizando dos banderas booleanas (flag[0] y flag[1]), una para cada proceso.

Características:

- Cada proceso indica su intención de entrar a la sección crítica estableciendo su bandera en true.
- Antes de entrar, un proceso revisa la bandera del otro. Si la bandera del otro proceso es false, puede entrar.

Problema con la versión 2:

Esta versión no garantiza la exclusión mutua. Un proceso podría ser interrumpido justo después de establecer su bandera pero antes de verificar la del otro. Por ejemplo si P0 establece flag[0] = true, pero entonces P0 es interrumpido por el planificador de procesos y entonces llega P1 y establece que flag[1] = true y verifica que también flag[0] = true y entonces espera y cuando se reanuda el proceso de P0 y verifica que flag[1] = true entonces de igual manera se quedará esperando.

Versión 3. Banderas con Espera Activa

Para corregir la falla anterior, esta versión modifica el orden, es decir, un proceso primero establece su bandera y luego verifica la del otro.

Características:

- P0 establece flag[0] = true.
- Mientras flag[1] sea true, P0 espera.
- Cuando flag[1] se vuelve false, P0 entra a la sección crítica.

Algoritmo de Dekker y Peterson

- Al salir, P0 establece $\text{flag}[0] = \text{false}$.
- P1 hace lo análogo.

Problema con la versión 3:

Esta versión soluciona el problema de la exclusión mutua, pero introduce la posibilidad de un bloqueo mutuo. Si ambos procesos establecen sus banderas a true aproximadamente al mismo tiempo, ocurrirá lo siguiente; P0 establece $\text{flag}[0] = \text{true}$ y P1 establece $\text{flag}[1] = \text{true}$, entonces P0 verifica $\text{flag}[1]$ y, como es true, entra en un bucle de espera y P1 verifica $\text{flag}[0]$ y, como es true, también entra en un bucle de espera, por lo tanto, ambos procesos quedarán esperando indefinidamente a que el otro baje su bandera, algo que nunca sucederá.

Versión 4. Retirada Temporal de la Intención

Esta versión intenta resolver el bloqueo mutuo haciendo que un proceso retire su intención de entrar si detecta que el otro también quiere hacerlo.

Características:

- P0 establece $\text{flag}[0] = \text{true}$.
- Si $\text{flag}[1]$ es true, P0 retira su intención estableciendo $\text{flag}[0] = \text{false}$ por un breve momento y luego lo vuelve a intentar.

Problema con la versión 4:

Aunque esta solución evita el bloqueo mutuo, puede llevar a una inanición o postergación indefinida. Podría darse el caso de que alguno de los dos procesos esté constantemente cediendo el paso al otro proceso y que por esta misma razón nunca consiga entrar, porque de igual forma pueden surgir interrupciones inoportunas por parte del planificador de procesos.

La Solución Final (Versión 5): Combinando Banderas y Turnos (Funcionamiento detallado)

En esta última versión se cubren todos los posibles problemas que pudieran llegar a surgir porque se basa en una combinación de banderas y turnos.

Características:

- Utiliza tanto las banderas ($\text{flag}[0]$, $\text{flag}[1]$) para indicar la intención como una variable turn para resolver conflictos.
- P0, establece su bandera $\text{flag}[0] = \text{true}$.
- Luego, verifica la bandera del otro proceso, P1.
 - Si $\text{flag}[1]$ es false, P0 puede entrar a la sección crítica directamente.
 - Si $\text{flag}[1]$ es true, significa que hay un conflicto, en este caso, se consulta la variable turn.
 - ❖ Si turn es 0, P0 insiste y sigue esperando a que P1 baje su bandera.
 - ❖ Si turn es 1, P0 sabe que es el turno de P1, por lo que retira su intención ($\text{flag}[0] = \text{false}$), espera a que turn sea 0, y luego vuelve a establecer su bandera e intenta de nuevo.

- Al salir de la sección crítica, el proceso cede el turno al otro (turn = 1) y baja su bandera (flag[0] = false).

Pseudocódigo:

Para el Proceso i

hacer

```
{
    // Sección de Entrada
    quiere_entrar[i] = verdadero;
    mientras (quiere_entrar[j])
    {
        si (turno == j)
        {
            quiere_entrar[i] = falso;
            mientras (turno == j)
            {
                // esperar
            }
            quiere_entrar[i] = verdadero;
        }
    }

    // Sección Crítica
    // ... (código a ejecutar en exclusión mutua)

    // Sección de Salida
    turno = j;
    quiere_entrar[i] = falso;

    // Sección Restante
    // ...

} mientras (verdadero);
```

Demostración de Correctitud (Las tres condiciones)

Como último punto podemos defender que la quinta y última versión del algoritmo de Dekker cumple con los 3 requisitos para la solución del problema de la sección crítica.

1. Exclusión Mutua: Es imposible que ambos procesos estén en la sección crítica a la vez. Si ambos lo intentan, la variable *turn* actúa como un árbitro infalible que solo deja pasar a uno, mientras el otro se ve forzado a esperar.
2. Progreso (Sin Bloqueo Mutuo o Deadlock): Si un proceso quiere entrar y el otro no, el primero podrá hacerlo sin quedar bloqueado. Si ambos quieren entrar, el mecanismo de *turn* asegura que uno de los dos avance, evitando que se queden ambos esperando indefinidamente.
3. Espera Limitada (Sin Inanición o Starvation): Gracias a que la variable *turn* se cede obligatoriamente al salir, un proceso no puede entrar en la sección crítica repetidamente mientras el otro espera. Se garantiza que los procesos se alternarán si ambos tienen una demanda continua de acceso, evitando que uno monopolice el recurso.

3. El Algoritmo de Peterson: Una Simplificación Elegante

3.1 Origen y Motivación

El Algoritmo de Peterson (1981) es una solución clásica al problema de la sección crítica, donde varios procesos compiten por acceder a recursos compartidos. Garantiza que solo un proceso esté en la sección crítica a la vez, evitando condiciones de carrera y asegurando la integridad de los datos. Es una versión simplificada del Algoritmo de Dekker y funciona completamente mediante software, usando solo memoria compartida para la comunicación entre procesos.

3.2 El Algoritmo para Dos Procesos (Funcionamiento detallado)

La versión original del Algoritmo de Peterson está diseñada para dos procesos, P_0 y P_1 , y utiliza dos variables globales compartidas:

flag: Un arreglo booleano de dos elementos, $flag[0]$ y $flag[1]$. Si $flag[i] = true$, indica que el proceso P_i desea entrar en su sección crítica.

turn: Una variable entera que indica a cuál proceso se le concede prioridad en caso de conflicto.

Funcionamiento paso a paso

El protocolo de Peterson se basa en la cortesía mutua. Para que un proceso P_i (donde j es el otro proceso, $j = 1 - i$) entre en su sección crítica, sigue estos pasos :

Expresión de la intención:

El proceso P_i establece su bandera a *true*, declarando su deseo de entrar en la sección crítica.

$flag[i] = true;$

Cesión de la prioridad: P_i establece la variable **turn** al identificador del otro proceso, j , cediéndole la prioridad en caso de una contienda.

$turn = j;$

Condición de espera:

P_i entra en un bucle while y se queda esperando si el otro proceso P_j también quiere entrar ($flag[j] == true$) Y es el turno de P_i ($turn == j$). Si alguna de estas condiciones es falsa, el bucle termina y P_i puede entrar a su sección crítica.

$while (flag[j] \&\& turn == j);$

Liberación del recurso:

Al salir de la sección crítica, P_i restablece su bandera a *false*, señalando que ha terminado y permitiendo que otros procesos puedan proceder.

$flag[i] = false;$

Pseudocódigo de P_i :

C/C++

```
// Variables compartidas
bool flag[2] = {false, false};
int turn;

// Código del proceso Pi
do {
```

```
// Sección de entrada
flag[i] = true;
turn = j;
while (flag[j] && turn == j);

// Sección crítica

// Sección de salida
flag[i] = false;

// Sección remanente
} while (true);
```

Ahora bien, en el funcionamiento anterior supusimos que P_j no quería entrar a la sección crítica. Pero el verdadero propósito del algoritmo es solucionar cuando ambos procesos intentan acceder de manera simultánea a su sección crítica

Expresión de la intención:

Los procesos P_i y P_j establecen su bandera a *true*, indicando que desean entrar en la sección crítica.

$flag[i] = true;$, $flag[j] = true;$,

Cesión de la prioridad: Cada proceso cede la prioridad al otro

$P_i: turn = j$, $P_j: turn = i$

En esta asignaciones es donde se encuentra el detalle clave del algoritmo. **turn** es una sola variable compartida, así que solo puede tener un valor en un instante. Aunque ambos procesos ejecuten la asignación “al mismo tiempo”, en la práctica, uno de los dos escribirá al último en la variable, debido a que la memoria se actualiza en orden físico. Por lo tanto, **turn** va a tomar el último valor que haya sido escrito en él, determinando así cuál proceso es el que entrará primero a la sección crítica.

En esta ocasión, decimos que el último valor en escribir fue $turn = j$

Condición de espera:

P_i y P_j entran en un bucle while y ambas evalúan sus condiciones:

Para P_i : $while(flag[j] \&\& turn == j)$

En P_i evaluamos si el proceso P_j también quiere entrar a la sección crítica, lo cual es cierto, después, evaluamos si es el turno de P_j , que también es correcto, por lo cual el bucle se cumple, manteniendo a P_i en espera, ejecutando el mismo bucle una y otra vez

Para P_j : $while(flag[i] \&\& turn == i)$

En P_j , evaluamos si el proceso P_i también quiere entrar a la sección crítica, lo cual es cierto, ahora evaluamos si es el turno de i , sin embargo, sabemos $turn$ se quedó con el valor de j , por lo cual, esta condición no se cumple y el proceso P_j sale del bucle, accediendo así a la sección crítica, mientras P_i sigue en espera bloqueado en el bucle while

Liberación del recurso:

Al salir de la sección crítica, P_j restablece su bandera a *false*, $flag[i] = false$; señalando que ha terminado y permitiendo que la condición $flag[i]$ deje de cumplir para P_i , por lo que ahora P_i puede acceder a la sección crítica.

3.3 Demostración de Correctitud (exclusión mutua, progreso, espera limitada)

Una solución válida al problema de la sección crítica debe satisfacer tres requisitos fundamentales. El Algoritmo de Peterson cumple los tres, lo que lo convierte en una solución formalmente correcta :

Exclusión Mutua: Esta propiedad garantiza que solo un proceso a la vez pueda estar en la sección crítica. Se demuestra por contradicción: si ambos procesos estuvieran en sus secciones críticas, entonces $turn$ debería ser igual a 0 y 1 al mismo tiempo, lo cual es lógicamente imposible.

Progreso: Si ningún proceso está en su sección crítica y uno o más procesos quieren entrar, solo aquellos que están esperando pueden decidir quién entra a continuación. El algoritmo de Peterson satisface esta condición porque un proceso que no quiere entrar ($flag[j] = false$) no puede bloquear a otro.

Espera Limitada (Bounded Waiting): Esta propiedad asegura que un proceso que desea entrar a la sección crítica no espere indefinidamente, lo que se conoce como inanición. La lógica de cesión de turno de Peterson garantiza que un proceso que espera solo lo hará durante un ciclo del otro proceso antes de que se le conceda la entrada.

3.4 Generalización para N Procesos (Lógica y funcionamiento)

El algoritmo de Peterson original funciona solo para dos procesos. Para extenderlo a N procesos, se necesita una versión iterativa en la que cada proceso avance por varias etapas o niveles, compitiendo en cada uno hasta que solo uno logre acceder a la sección crítica. Este algoritmo utiliza dos arrays compartidos para gestionar la sincronización:

- **nivel[N]**: Un array de enteros que indica el nivel actual de cada proceso en la "competencia". Inicialmente, todos los valores son 0.
- **victima[N-1]**: Un array de enteros que, para cada nivel, almacena el último proceso que llegó a dicho nivel. El proceso víctima de cada nivel es el que menos prioridad tiene de subir

Para poder comprender el funcionamiento de este algoritmo emplearemos un ejemplo breve

Supongamos que 3 procesos quieran entrar a la sección crítica (P0,P1, P2).

Las variables compartidas se verán así inicialmente:

nivel[3] = { 0, 0, 0 }

victima[2] = {?, ? }

Condición de Espera:

while (existe q != p tal que level[q] >= i+1 Y victim[i] == p)

None

```
// Bucle para subir de nivel
```

```
for i from 0 to N-1 {
```

```
    // Declara tu intención y nivel
```

```
    level[p] = i;
```

```
    // Te designas como la "víctima" para este nivel
```

```
    victim[i] = p;
```

```
    // Bucle de espera ocupada
```

```
    while ( (existe q != p tal que level[q] >= i) AND (victim[i] == p) ) {
```

```
        // Espera
```

```
    }
```

```
}
```

Iteramos sobre i

Nivel 1 ($i = 0$)

Los tres procesos intentan avanzar al nivel 1

$P0$ establece $nivel[0] = 1$, $victima[0] = 0$

$P1$ establece $nivel[1] = 1$, $victima[0] = 1$

$P2$ establece $nivel[2] = 1$, $victima[0] = 2$

El último en modificar $victima[0]$ es $P2$, por lo que el valor final es 2. Esto significa que $P2$ es la víctima para este nivel, y debe esperar en él, mientras que los otros procesos pueden continuar

$P0$ y $P1$ entran en su bucle de espera,

while (*existe $q \neq p$ tal que $level[q] \geq 1$ Y $victima[0] == p$*), pero $victima[0] = 2$, por lo que la condición es falsa para $P0$ y $P1$, lo que permite que estos avancen al siguiente nivel

Como la víctima es $P2$, su condición es verdadera y tiene que quedarse en espera en el nivel 1

Nivel 2 ($i=1$)

Ahora, $P0$ y $P1$ compiten. $P2$ sigue esperando en el nivel 1.

$P0$ establece $nivel[0] = 2$, $victima[2] = 0$

$P1$ establece $nivel[1] = 2$, $victima[2] = 1$

Ahora, en el nivel 2, $P1$ queda como la víctima

$P0$ evalúa su bucle de espera:

while (*existe $q = 1$ tal que $level[1] \geq 2$ Y $victim[1] == 0$*).

La segunda parte de la condición, $victim[1] == 0$, es falsa. $P0$ no espera. Pasa la prueba del bucle de espera y puede entrar a la sección crítica.

$P1$ evalúa su bucle de espera:

while (*existe $q = 0$ tal que $level[0] \geq 2$ Y $victim[1] == 1$*).

La segunda parte de la condición, $victim[1] == 1$, es verdadera. $P1$ espera.

Acceso a la Sección Crítica

P0 es el único proceso que ha superado todos los niveles. Por lo tanto, accede a la sección crítica. P1 y P2 están esperando.

Salida de la Sección Crítica

Cuando P0 sale, establece su nivel a 0: $nivel[0] = 0$.

Esto "libera" a P1 de su espera, ya que la condición $nivel[0] \geq 2$ ahora es falsa. P1 puede continuar su avance.

Eventualmente, P1 entrará y saldrá, lo que permitirá que P2 avance y así sucesivamente.

4. Análisis Comparativo y Limitaciones Fundamentales

- Comparación Directa: Lógica, Complejidad y Elegancia Dekker vs Peterson (Puede ser una tabla comparativa)

Algoritmo	Lógica	Complejidad	Elegancia
Dekker	Resuelve la exclusión mediante banderas de interés y una serie de chequeos adicionales que detectan conflicto y resuelven por alternancia (turn); usa varias comprobaciones para romper la simetría.	Conceptualmente más complejo: más pasos por entrada a la sección crítica; espacio $O(1)$ pero mayor "overhead" por la lógica multietapa.	Menos elegante: ingenioso históricamente pero poco conciso y difícil de razonar a primera vista.
Peterson	Dos banderas + turn: un proceso indica interés, cede el turno al otro y espera mientras el otro quiera entrar y sea su turno, tiene lógica simple y directa.	Menor complejidad conceptual y práctica: menos comprobaciones y operaciones por entrada; también $O(1)$ para 2 procesos (la generalización a N aumenta costos).	Más elegante y claro: corto, fácil de entender y demostrar (baja consistencia secuencial).

- La Ineficiencia de la Espera Activa (Busy-Waiting)

El Busy waiting es ineficiente por varias razones:

❖ **Desperdicio de CPU**

Un proceso que está en bucle leyendo una variable compartida consume ciclos de CPU en vano. En sistemas con tiempo de CPU escaso o con muchos hilos, esto termina siendo un problema.

❖ **Ping-pong de caché (cache-line bouncing)**

Si los hilos en diferentes núcleos leen o escriben la misma variable, la línea de caché salta entre núcleos (coherence traffic), degradando latencia y rendimiento. Esto agrava la contención.

❖ **Preemption y priorización**

Si el hilo que posee la sección crítica es preemptado (que un proceso fue interrumpido por el sistema operativo para darle la CPU a otro), los demás pueden seguir girando quemando CPU, lo que provoca degradación y, en la práctica, efectos similares a bloqueo.

❖ **Escalabilidad pobre**

Para más de unos pocos hilos/núcleos, spinning en variables compartidas no escala: más contención → más latencia → menos rendimiento.

❖ **Mitigaciones parciales**

- Usar instrucciones de pausa (PAUSE en x86) o yield/sched_yield para reducir el impacto.
- Backoff exponencial (esperas crecientes) reduce colisión pero no elimina overhead.
- Usar primitivas que combinen espera y bloqueo en el kernel para evitar spinning prolongado.
- Desafíos en Arquitecturas Modernas

● **Desafíos en arquitecturas modernas**○ **Modelos de memoria débiles**

En ARM, POWER y hasta x86, las operaciones pueden reordenarse. Sin *fences* o atómicos, Dekker y Peterson pierden sus garantías.

○ **Optimización del compilador**

El compilador puede reordenar o eliminar lecturas. Se necesitan std::atomic y órdenes de memoria correctas para mantener la lógica.

○ **Caches y false sharing**

Variables como flags y turn pueden compartir línea de caché y provocar *bouncing*,

reduciendo el rendimiento.

- **Escalabilidad limitada**

Están pensados para 2 procesos; al generalizarse a N, el costo crece y no escalan bien en sistemas con muchos hilos.

- **Preemption y prioridades**

Si el hilo con el lock es desplanificado, los demás gastan CPU en espera activa; los locks modernos evitan esto bloqueando al hilo.

5. Soluciones Modernas al Problema de la Sección Crítica

-Soluciones Basadas en Hardware: Instrucciones Atómicas

Estas soluciones aprovechan el hardware del procesador para ejecutar una operación como una sola, **indivisible unidad** (atómica). Esto significa que la operación no puede ser interrumpida. Dos ejemplos comunes son:

- **TestAndSet**: Esta instrucción lee una variable y, al mismo tiempo, establece su valor en **true**. Si la variable estaba en **false**, el proceso entra a la sección crítica. Si estaba en **true**, el proceso se queda esperando.
- **Swap**: Esta instrucción intercambia atómicamente el valor de dos variables. Un proceso podría usar **Swap** para intercambiar una variable local con una variable de bloqueo compartida. Si el valor original de la variable de bloqueo era **false**, el proceso puede entrar.

Ventaja: Son muy eficientes y rápidas, ya que la exclusión mutua se garantiza a nivel de hardware.

Desventaja: Pueden llevar a la **espera activa** (spinning), donde los procesos gastan tiempo de CPU revisando continuamente la variable de bloqueo, lo que es ineficiente si el recurso está ocupado por mucho tiempo.

-Primitivas del Sistema Operativo: Evitando la Espera Activa

Para evitar la espera activa, los sistemas operativos ofrecen herramientas que permiten que un proceso que no puede entrar a la sección crítica **se duerma** (bloquee) en lugar de consumir ciclos de CPU. Cuando el recurso está libre, el sistema operativo lo "despierta" (lo pone en la cola de listos para ejecutarse).

- **Mutexes (Mutual Exclusion)**: Son la forma más simple de bloqueo. Un mutex tiene dos estados: **bloqueado** y **desbloqueado**. Un proceso que quiere entrar a la sección crítica lo bloquea (**lock**). Cuando termina, lo desbloquea (**unlock**). Si otro proceso intenta bloquearlo cuando ya está bloqueado, se duerme.
- **Semáforos**: Son más versátiles que los mutexes. Un semáforo es una variable entera que se manipula con dos operaciones atómicas:

- **wait()** (o **P**): Disminuye el valor del semáforo. Si el valor es negativo, el proceso se bloquea.
- **signal()** (o **V**): Aumenta el valor del semáforo. Si hay procesos bloqueados, uno de ellos es despertado.
- Los semáforos se usan para controlar el acceso a un número limitado de recursos (semáforo de conteo) o para exclusión mutua (semáforo binario, que es equivalente a un mutex).

-Soluciones de Software Avanzadas: Algoritmo de la Panadería de Lamport

El **Algoritmo de la Panadería de Lamport** es una solución de software que garantiza la exclusión mutua para **N procesos**. Se basa en una analogía con una panadería: antes de ser atendido, un cliente toma un número. A los clientes se les atiende en orden según su número, de menor a mayor.

Lógica y Funcionamiento Detallado

Cada proceso, antes de entrar a la sección crítica, pasa por dos fases:

1. Fase de Toma de Número (Choosing):

- Cada proceso **i** tiene una variable booleana **choosing[i]**, que inicialmente es **false**.
- Un proceso que quiere un número establece **choosing[i] = true**.
- Luego, encuentra el número más alto que ya ha sido asignado a otros procesos y toma un número (**number[i]**) que es uno más que el más alto encontrado.
- Una vez que tiene su número, establece **choosing[i] = false**.
- La fase **choosing** es crucial para que dos procesos no tomen el mismo número simultáneamente.

2. Fase de Espera:

- Una vez que el proceso **i** tiene su número, debe esperar a que los otros procesos sean atendidos.
- Recorre todos los demás procesos **j** (donde **j** es diferente de **i**).
- Para cada proceso **j**, espera hasta que **choosing[j]** sea **false** (lo que significa que **j** ya tiene su número).
- Luego, espera hasta que se cumpla la siguiente condición: (**number[j], j**) no es menor que (**number[i], i**).
- Esta condición es la regla de atención: se atiende al cliente con el número más bajo. En caso de que dos procesos tengan el mismo número, se usa el ID del proceso (**i** o **j**) como desempate, atendiendo al de menor ID.

Cuando estas dos condiciones se cumplen para todos los demás procesos, el proceso **i** puede entrar a su sección crítica.

Ventajas:

- Garantiza la exclusión mutua.

- Evita el interbloqueo (deadlock) y la inanición (starvation).
- No requiere hardware especial y funciona para N procesos.

Desventajas:

- Requiere espera activa (spinning), ya que los procesos revisan continuamente las variables de los demás.
- La implementación es compleja.
- Es menos eficiente que las soluciones basadas en hardware o primitivas del SO en entornos donde la exclusión mutua no es frecuente.

6. Conclusión: El Legado de Dekker y Peterson en la Computación Concurrente

● Síntesis del Análisis

El problema de la sección crítica surge de la necesidad de sincronizar procesos que comparten recursos no compartibles. En su momento, los algoritmos de Dekker y Peterson representaron avances cruciales: Dekker fue la primera solución puramente de software que garantizó exclusión mutua, progreso y espera limitada; mientras que Peterson simplificó la lógica, ofreciendo un método más claro y elegante para dos procesos, y que incluso sirvió como base para generalizaciones posteriores. Ambos enfoques evidenciaron la importancia de contar con protocolos de entrada y salida que prevengan condiciones de carrera, inanición o bloqueos mutuos.

● Importancia y Legado

El aporte de estos algoritmos va más allá de su uso práctico: constituyen un hito histórico en la evolución de la programación concurrente. Aunque hoy en día las soluciones modernas descansan en primitivas de hardware (instrucciones atómicas) y en mecanismos provistos por el sistema operativo (mutexes, semáforos, monitores), el legado de Dekker y Peterson se mantiene como fundamento teórico y pedagógico. Su estudio permite comprender los principios de exclusión mutua, progreso y equidad, que siguen vigentes en las técnicas actuales de sincronización. Además, impulsaron la transición del razonamiento informal a la formalización matemática de algoritmos concurrentes, abriendo paso a modelos más sofisticados como el algoritmo de la panadería de Lamport o las soluciones basadas en memoria compartida atómica.

● Conclusión Final

El trabajo de Dekker y Peterson no debe verse únicamente como soluciones obsoletas, sino como la raíz de todo el campo de la concurrencia moderna. Su importancia radica en haber sentado las bases conceptuales para entender la sincronización, la coordinación entre procesos y los retos del paralelismo. En un mundo donde los sistemas multiprocesador y la computación distribuida son la

norma, sus algoritmos siguen siendo un recordatorio de que la simplicidad y la corrección formal son esenciales para construir software confiable. En resumen, el legado de estos pioneros es haber transformado un problema técnico en un campo completo de estudio, sobre el cual se sigue edificando la computación actual.

7. Bibliografía

- UNJu Virtual. (s.f.). TEMA -3. Recuperado de <https://virtual.unju.edu.ar/mod/resource/view.php?id=230815>
- Microsoft Learn. (s.f.). Introducción a las primitivas de sincronización. Recuperado de <https://learn.microsoft.com/es-es/dotnet/standard/threading/overview-of-synchronization-primitives>
- UNICAN - Universidad de Cantabria. (s.f.). PROGRAMACION CONCURRENTES. Recuperado de https://www.ctr.unican.es/asignaturas/procodis_3_ii/doc/procodis_2_03.pdf
- Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem. Communications of the ACM, 17(8), 453-455. <https://doi.org/10.1145/361082.361093>
- Geeks for Geeks (2023) Algoritmo de Peterson para n procesos. Geeks for Geeks. Recuperado de: <https://www.geeksforgeeks.org/operating-systems/n-process-peterson-algorithm/>
- Tutorialspoint (2023) Algoritmo de Peterson en la sincronización de procesos. Tutorialspoint. Recuperado de: <https://www.tutorialspoint.com/petersons-algorithm-in-process-synchronization>
- Ted Mconel. (2022) Algoritmo de Peterson. [Video de YouTube]. Recuperado de: <https://youtu.be/E7Bzep1euFI?si=qpHG4L44z6jYe-up>
- Facultad de Ingeniería, Universidad de la República. (s.f.). Sistemas operativos: Concurrencia. Recuperado de: <https://www.fing.edu.uy/tecnoinf/mvd/cursos/so/material/teo/so07-concurrencia.pdf>
- Pérez, M. A., Tazon, S., Campo, J. C., & Drake, J. M. (s.f.). Tema 2: Sincronización de procesos. Universidad de Cantabria, Departamento de Electrónica y Computadores. Recuperado de: https://www.ctr.unican.es/asignaturas/procodis_3_ii/doc/procodis_2_03.pdf
- Equipo de Ciberseguridad.com. (s.f.). ¿Qué es una condición de carrera? (Race Condition). Ciberseguridad.com. Recuperado de: <https://ciberseguridad.com/amenazas/vulnerabilidades/condicion-de-carrera/>

Algoritmo de Dekker y Peterson

- Dos Santos, G. (s.f.). Sincronización: Problemas clásicos. Departamento de Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur. Recuperado de: <https://cs.uns.edu.ar/~gd/soyd/clases/04-SincronizacionProbClasicos.pdf>
- Morales, H. S. (2017). Unidad 4: Exclusión mutua y sincronización. Facultad de Ciencias de la Computación, Benemérita Universidad Autónoma de Puebla. Recuperado de: https://www.cs.buap.mx/~hilario_sm/slide/pcp2016/unidad%204%20pcp-2017.pdf
- Buhr, P. A., Dice, D., & Hesselink, W. H. (2015). Dekker's mutual exclusion algorithm made RW-safe. Concurrency and Computation: Practice & Experience, 28(1), 144-165. <https://doi.org/10.1002/cpe.3659>
- What Does "Busy Waiting" Mean in Operating Systems? | Baeldung on Computer Science. (s. f.). Recuperado de <https://www.baeldung.com/cs/os-busy-waiting>