

# Programación Concurrente

Ing. Lamberto Maza Casas

Marzo, 2013

## Programación Concurrente

Es el nombre dado a la notación y técnicas de programación utilizados para expresar paralelismo potencial y resolver los problemas de sincronización y comunicación resultantes ([1]).

## Programación Concurrente

Es el nombre dado a la notación y técnicas de programación utilizados para expresar paralelismo potencial y resolver los problemas de sincronización y comunicación resultantes ([1]).

## Proceso

Todos los lenguajes con los que se hace programación concurrente incorporan implícita o explícitamente, la noción de *proceso*; cada proceso tiene un solo hilo de control.

## Programa concurrente

De acuerdo con [2], un programa concurrente se ve convencionalmente como una colección de procesos secuenciales autónomos, ejecutándose (lógicamente) en paralelo.

## Programa concurrente

De acuerdo con [2], un programa concurrente se ve convencionalmente como una colección de procesos secuenciales autónomos, ejecutándose (lógicamente) en paralelo.

## Implementación

La implementación real (ejecución) de una colección de procesos usualmente toma una de las siguientes tres formas ([1]). Los procesos pueden

Son posibles formas híbridas de los tres métodos.

## Programa concurrente

De acuerdo con [2], un programa concurrente se ve convencionalmente como una colección de procesos secuenciales autónomos, ejecutándose (lógicamente) en paralelo.

## Implementación

La implementación real (ejecución) de una colección de procesos usualmente toma una de las siguientes tres formas ([1]). Los procesos pueden

- ▶ multiplexar sus ejecuciones sobre un único procesador,

Son posibles formas híbridas de los tres métodos.

## Programa concurrente

De acuerdo con [2], un programa concurrente se ve convencionalmente como una colección de procesos secuenciales autónomos, ejecutándose (lógicamente) en paralelo.

## Implementación

La implementación real (ejecución) de una colección de procesos usualmente toma una de las siguientes tres formas ([1]). Los procesos pueden

- ▶ multiplexar sus ejecuciones sobre un único procesador,
- ▶ multiplexar sus ejecuciones sobre un sistema multiprocesador donde hay acceso a memoria compartida,

Son posibles formas híbridas de los tres métodos.

## Programa concurrente

De acuerdo con [2], un programa concurrente se ve convencionalmente como una colección de procesos secuenciales autónomos, ejecutándose (lógicamente) en paralelo.

## Implementación

La implementación real (ejecución) de una colección de procesos usualmente toma una de las siguientes tres formas ([1]). Los procesos pueden

- ▶ multiplexar sus ejecuciones sobre un único procesador,
- ▶ multiplexar sus ejecuciones sobre un sistema multiprocesador donde hay acceso a memoria compartida,
- ▶ multiplexar sus ejecuciones sobre varios procesadores los cuales no comparten memoria (tales sistemas son usualmente llamados sistemas distribuidos).

Son posibles formas híbridas de los tres métodos.



## Procesos en Sistemas Operativos

El concepto central de cualquier sistema operativo es el *proceso*: una abstracción de un programa en ejecución.

## Procesos en Sistemas Operativos

El concepto central de cualquier sistema operativo es el *proceso*: una abstracción de un programa en ejecución.

### Programa ejecutable

La estructura de todo programa ejecutable creado por el compilador de C (véase [3]) viene impuesta por el sistema. En Linux un programa consta de las siguientes partes:

## Procesos en Sistemas Operativos

El concepto central de cualquier sistema operativo es el *proceso*: una abstracción de un programa en ejecución.

### Programa ejecutable

La estructura de todo programa ejecutable creado por el compilador de C (véase [3]) viene impuesta por el sistema. En Linux un programa consta de las siguientes partes:

- ▶ Un conjunto de cabeceras que describen atributos del fichero.

## Procesos en Sistemas Operativos

El concepto central de cualquier sistema operativo es el *proceso*: una abstracción de un programa en ejecución.

### Programa ejecutable

La estructura de todo programa ejecutable creado por el compilador de C (véase [3]) viene impuesta por el sistema. En Linux un programa consta de las siguientes partes:

- ▶ Un conjunto de cabeceras que describen atributos del fichero.
- ▶ Un bloque donde se encuentran las instrucciones en lenguaje máquina del programa. Este bloque se conoce en Unix como texto del programa.

## Procesos en Sistemas Operativos

El concepto central de cualquier sistema operativo es el *proceso*: una abstracción de un programa en ejecución.

### Programa ejecutable

La estructura de todo programa ejecutable creado por el compilador de C (véase [3]) viene impuesta por el sistema. En Linux un programa consta de las siguientes partes:

- ▶ Un conjunto de cabeceras que describen atributos del fichero.
- ▶ Un bloque donde se encuentran las instrucciones en lenguaje máquina del programa. Este bloque se conoce en Unix como texto del programa.
- ▶ Un bloque dedicado a la representación en lenguaje máquina de los datos que deben ser inicializados cuando arranca la ejecución del programa. Aquí está incluida una indicación de cuánto espacio de memoria debe reservar el núcleo para estos datos.

Tradicionalmente, este bloque se conoce como bss —pseudo-operador del ensamblador del IBM 7090 que significa *block started by symbol*—. El núcleo inicializa, en tiempo de ejecución, esta zona a valor 0.

Tradicionalmente, este bloque se conoce como bss —pseudo-operador del ensamblador del IBM 7090 que significa *block started by symbol*—. El núcleo inicializa, en tiempo de ejecución, esta zona a valor 0.

- ▶ Otras secciones, tales como tablas de símbolos.

Tradicionalmente, este bloque se conoce como bss —pseudo-operador del ensamblador del IBM 7090 que significa *block started by symbol*—. El núcleo inicializa, en tiempo de ejecución, esta zona a valor 0.

- ▶ Otras secciones, tales como tablas de símbolos.

## Proceso

Cuando un programa es leído del disco por el núcleo y es cargado en memoria para ejecutarse se convierte en un proceso.



Tradicionalmente, este bloque se conoce como bss —pseudo-operador del ensamblador del IBM 7090 que significa *block started by symbol*—. El núcleo inicializa, en tiempo de ejecución, esta zona a valor 0.

- ▶ Otras secciones, tales como tablas de símbolos.

## Proceso

Cuando un programa es leído del disco por el núcleo y es cargado en memoria para ejecutarse se convierte en un proceso.

En un proceso no sólo hay una copia del programa, también nos encontramos información de control añadida por el núcleo.

## Segmentos de proceso

Un proceso se compone de tres bloques fundamentales conocidos como segmentos ([3]):

## Segmentos de proceso

Un proceso se compone de tres bloques fundamentales conocidos como segmentos ([3]):

- ▶ El *segmento de texto* contiene las instrucciones que entiende la CPU de nuestra máquina. Este bloque es una copia del bloque de texto del programa.

## Segmentos de proceso

Un proceso se compone de tres bloques fundamentales conocidos como segmentos ([3]):

- ▶ El *segmento de texto* contiene las instrucciones que entiende la CPU de nuestra máquina. Este bloque es una copia del bloque de texto del programa.
- ▶ El *segmento de datos* contiene los datos que deben ser inicializados al arrancar el proceso. Si el programa ha sido generado por un compilador de C, en este bloque estarán las variables globales y estáticas. Se corresponde con el bloque *bss* del programa.

## Segmentos de proceso

Un proceso se compone de tres bloques fundamentales conocidos como segmentos ([3]):

- ▶ El *segmento de texto* contiene las instrucciones que entiende la CPU de nuestra máquina. Este bloque es una copia del bloque de texto del programa.
- ▶ El *segmento de datos* contiene los datos que deben ser inicializados al arrancar el proceso. Si el programa ha sido generado por un compilador de C, en este bloque estarán las variables globales y estáticas. Se corresponde con el bloque *bss* del programa.
- ▶ El *segmento de pila*. Lo crea el núcleo al arrancar el proceso y su tamaño es gestionado dinámicamente por el núcleo. La pila se compone de una serie de bloques lógicos, llamados *marcos de pila*, que son introducidos cuando se llama a una función y son sacados cuando se vuelve de la función.

## Marco de pila

Un marco de pila se compone de los parámetros de la función, las variables locales de la función y la información necesaria para restaurar el marco de pila anterior a la llamada a la función —dentro de esta información se incluyen el contador de programa y el apuntador de pila anteriores a la llamada a la función—.

## Marco de pila

Un marco de pila se compone de los parámetros de la función, las variables locales de la función y la información necesaria para restaurar el marco de pila anterior a la llamada a la función —dentro de esta información se incluyen el contador de programa y el apuntador de pila anteriores a la llamada a la función—.

En los programas fuente no se incluye código para gestionar la pila —a menos que estén escritos en ensamblador—; es el compilador quien incluye el código necesario para controlarla.

## Modos de ejecución de procesos

Los procesos se ejecutan en dos modos: usuario y supervisor —este último es conocido también como modo kernel—; el sistema maneja sendas pilas por separado. La pila del modo usuario contiene los argumentos, variables locales y otros datos relativos a funciones que se ejecutan en modo usuario, y la pila del modo supervisor contiene los marcos de pila de las llamadas al sistema —estas se ejecutan en modo supervisor—.



## Modos de ejecución de procesos

Los procesos se ejecutan en dos modos: usuario y supervisor —este último es conocido también como modo kernel—; el sistema maneja sendas pilas por separado. La pila del modo usuario contiene los argumentos, variables locales y otros datos relativos a funciones que se ejecutan en modo usuario, y la pila del modo supervisor contiene los marcos de pila de las llamadas al sistema —estas se ejecutan en modo supervisor—.

Unix es un sistema de tiempo compartido que permite multiproceso —ejecución de varios procesos concurrentemente—. El planificador (scheduler) es la parte del núcleo encargada de gestionar la CPU y determinar qué proceso pasa a ocupar la CPU en un determinado instante. Un mismo programa puede estar siendo ejecutado en un instante determinado por varios procesos a la vez.

## ¿Cómo se crean los procesos?

Desde un punto de vista funcional, un proceso en Unix es una entidad creada tras la llamada **fork**. Todos los procesos, excepto el primero —proceso número 0—, son creados mediante una llamada a `fork`. el proceso que llama a `fork` se conoce como proceso padre y el proceso creado es el proceso hijo. Todos los procesos tienen un único proceso padre, pero pueden tener varios procesos hijos.

## ¿Cómo se crean los procesos?

Desde un punto de vista funcional, un proceso en Unix es una entidad creada tras la llamada **fork**. Todos los procesos, excepto el primero —proceso número 0—, son creados mediante una llamada a `fork`. el proceso que llama a `fork` se conoce como proceso padre y el proceso creado es el proceso hijo. Todos los procesos tienen un único proceso padre, pero pueden tener varios procesos hijos.

## Identificador de Proceso

El núcleo identifica cada proceso mediante su PID —*process identification*—, que es un número asociado a cada proceso y que no cambia durante el tiempo de vida de éste.

El proceso 0 es especial: es creado cuando arranca el sistema, y después de hacer una llamada a `fork` se convierte en el proceso intercambiador —encargado de la gestión de la memoria virtual—. El proceso hijo creado se llama **init** y su PID vale 1. Este proceso es el encargado de arrancar los demás procesos del sistema según la configuración que se indica en el fichero `/etc/inittab`.

## ¿Qué es realmente un proceso?

Un **proceso** es una instancia de un programa en ejecución y también la unidad básica de planificación de Linux ([4]).

## ¿Qué es realmente un proceso?

Un **proceso** es una instancia de un programa en ejecución y también la unidad básica de planificación de Linux ([4]).

El núcleo utiliza procesos para controlar el acceso a la CPU y a otros recursos del sistema y para determinar qué programas se ejecutan en la CPU, por cuánto tiempo, y con qué características.

## ¿Qué es realmente un proceso?

Un **proceso** es una instancia de un programa en ejecución y también la unidad básica de planificación de Linux ([4]).

El núcleo utiliza procesos para controlar el acceso a la CPU y a otros recursos del sistema y para determinar qué programas se ejecutan en la CPU, por cuánto tiempo, y con qué características.

El planificador del núcleo asigna el tiempo de ejecución de CPU, llamado *porciones de tiempo*, entre todos los procesos, vaciando de antemano cada proceso cuando termina su porción de tiempo.

## ¿Qué es realmente un proceso?

Un **proceso** es una instancia de un programa en ejecución y también la unidad básica de planificación de Linux ([4]).

El núcleo utiliza procesos para controlar el acceso a la CPU y a otros recursos del sistema y para determinar qué programas se ejecutan en la CPU, por cuánto tiempo, y con qué características.

El planificador del núcleo asigna el tiempo de ejecución de CPU, llamado *porciones de tiempo*, entre todos los procesos, vaciando de antemano cada proceso cuando termina su porción de tiempo.

Las porciones de tiempo son lo suficientemente pequeñas para que, en un sistema con un solo procesador, parezca que se están ejecutando simultáneamente varios procesos.



Cada proceso contiene también información sobre sí mismo, así que el núcleo puede activar o desactivar su ejecución según sea necesario.

Cada proceso contiene también información sobre sí mismo, así que el núcleo puede activar o desactivar su ejecución según sea necesario.

## Atributos de proceso

Los procesos también tienen un cierto número de atributos o características que los identifican y definen su comportamiento.

Cada proceso contiene también información sobre sí mismo, así que el núcleo puede activar o desactivar su ejecución según sea necesario.

## Atributos de proceso

Los procesos también tienen un cierto número de atributos o características que los identifican y definen su comportamiento.

El núcleo también mantiene internamente gran cantidad de información sobre cada proceso y proporciona una **interfaz** para obtenerla.

## PID y PPID

Los atributos mejor conocidos de un proceso son su ID (PID) y su ID de proceso padre (PPID). Ambos son enteros positivos. Un PID identifica de forma única a un proceso.

## PID y PPID

Los atributos mejor conocidos de un proceso son su ID (PID) y su ID de proceso padre (PPID). Ambos son enteros positivos. Un PID identifica de forma única a un proceso.

### ¿Por qué tiene que conocer un proceso su PID y PPID?

Una utilización común de un PID es la creación de nombres de archivo y directorios únicos. Por ejemplo, después de una llamada a **getpid**, el proceso podría utilizar el PID para crear un archivo temporal.

## PID y PPID

Los atributos mejor conocidos de un proceso son su ID (PID) y su ID de proceso padre (PPID). Ambos son enteros positivos. Un PID identifica de forma única a un proceso.

### ¿Por qué tiene que conocer un proceso su PID y PPID?

Una utilización común de un PID es la creación de nombres de archivo y directorios únicos. Por ejemplo, después de una llamada a **getpid**, el proceso podría utilizar el PID para crear un archivo temporal.

### Ejemplo 13.1

Véase ejemplo prpids.c

Otra tarea típica es escribir el PID en un archivo de registro como parte de un mensaje de registro, dejando claro qué proceso lo escribió. Un proceso puede utilizar su PPID para enviar una señal u otro mensaje a su proceso padre.

Otra tarea típica es escribir el PID en un archivo de registro como parte de un mensaje de registro, dejando claro qué proceso lo escribió. Un proceso puede utilizar su PPID para enviar una señal u otro mensaje a su proceso padre.

## ID reales y efectivos

Además de su PID y su PPID, cada proceso tiene otros atributos identificativos, que se muestran a continuación.



## Sincronización y Comunicación

Las mayores dificultades con la programación concurrente surgen de la interacción de los procesos.

## Sincronización y Comunicación

Las mayores dificultades con la programación concurrente surgen de la interacción de los procesos.

El comportamiento correcto de un programa concurrente depende críticamente de la sincronización y comunicación entre procesos.

## Sincronización y Comunicación

Las mayores dificultades con la programación concurrente surgen de la interacción de los procesos.

El comportamiento correcto de un programa concurrente depende críticamente de la sincronización y comunicación entre procesos.

### Sincronización

En su sentido más amplio, sincronización es la satisfacción de restricciones de entrelazado de las acciones de diferentes procesos (por ejemplo, que una acción particular de un proceso suceda solo después de una acción específica realizada por otro proceso).

## Sincronización y Comunicación

Las mayores dificultades con la programación concurrente surgen de la interacción de los procesos.

El comportamiento correcto de un programa concurrente depende críticamente de la sincronización y comunicación entre procesos.

### Sincronización

En su sentido más amplio, sincronización es la satisfacción de restricciones de entrelazado de las acciones de diferentes procesos (por ejemplo, que una acción particular de un proceso suceda solo después de una acción específica realizada por otro proceso).

### Comunicación

Comunicación es el paso de información de un proceso a otro.

## Relación entre sincronización y comunicación

Los dos conceptos están ligados, dado que algunas formas de comunicación requieren sincronización, y la sincronización puede ser considerada como una comunicación sin contenido.

## Relación entre sincronización y comunicación

Los dos conceptos están ligados, dado que algunas formas de comunicación requieren sincronización, y la sincronización puede ser considerada como una comunicación sin contenido.

## Formas de comunicación entre procesos

Comunicación entre procesos { Variables compartidas  
Paso de mensajes

## Variables compartidas

Las variables compartidas son objetos a los que más de un proceso tienen acceso; la comunicación por lo tanto, puede proceder con cada proceso referenciando esas variables cuando sea apropiado.

## Variables compartidas

Las variables compartidas son objetos a los que más de un proceso tienen acceso; la comunicación por lo tanto, puede proceder con cada proceso referenciando esas variables cuando sea apropiado.

### Ejemplo de variables compartidas usando memoria compartida entre procesos

Véanse los ejemplos IPC/SharedMemory/shm.c y fork/ejemplo01.c.



## Variables compartidas

Las variables compartidas son objetos a los que más de un proceso tienen acceso; la comunicación por lo tanto, puede proceder con cada proceso referenciando esas variables cuando sea apropiado.

## Ejemplo de variables compartidas usando memoria compartida entre procesos

Véanse los ejemplos IPC/SharedMemory/shm.c y fork/ejemplo01.c.

## Paso de mensajes

El paso de mensajes involucra intercambio de datos explícito entre dos procesos por medio de un mensaje que pasa de un proceso a otro.

## Variables compartidas

Las variables compartidas son objetos a los que más de un proceso tienen acceso; la comunicación por lo tanto, puede proceder con cada proceso referenciando esas variables cuando sea apropiado.

## Ejemplo de variables compartidas usando memoria compartida entre procesos

Véanse los ejemplos `IPC/SharedMemory/shm.c` y `fork/ejemplo01.c`.

## Paso de mensajes

El paso de mensajes involucra intercambio de datos explícito entre dos procesos por medio de un mensaje que pasa de un proceso a otro.

La elección entre variables compartidas y paso de mensajes recae en los diseñadores del lenguaje o del sistema operativo.

Las variables compartidas son fáciles de soportar si hay memoria compartida entre los procesos. Si no es así, aun se pueden usar si el hardware incorpora un medio de comunicación.

Las variables compartidas son fáciles de soportar si hay memoria compartida entre los procesos. Si no es así, aun se pueden usar si el hardware incorpora un medio de comunicación.

Similarmente, una primitiva de paso de mensajes puede ser soportada a través de memoria compartida o una red de paso de mensajes física.

Aunque las variables compartidas aparecen como una forma directa de pasar información entre procesos, su uso sin restricción no es confiable y es inseguro debido a problemas de actualizaciones múltiples.

Aunque las variables compartidas aparecen como una forma directa de pasar información entre procesos, su uso sin restricción no es confiable y es inseguro debido a problemas de actualizaciones múltiples.

Considere dos procesos actualizando una variable compartida  $X$ , con la asignación

$$X = X + 1;$$

En la mayoría de los hardware esto no será ejecutado en una operación **indivisible** (atómica), sino que será implementada en tres instrucciones distintas:

Aunque las variables compartidas aparecen como una forma directa de pasar información entre procesos, su uso sin restricción no es confiable y es inseguro debido a problemas de actualizaciones múltiples.

Considere dos procesos actualizando una variable compartida  $X$ , con la asignación

$$X = X + 1;$$

En la mayoría de los hardware esto no será ejecutado en una operación **indivisible** (atómica), sino que será implementada en tres instrucciones distintas:

(1) cargar el valor de  $X$  en algún registro (o en la cima del stack);

Aunque las variables compartidas aparecen como una forma directa de pasar información entre procesos, su uso sin restricción no es confiable y es inseguro debido a problemas de actualizaciones múltiples.

Considere dos procesos actualizando una variable compartida  $X$ , con la asignación

$$X = X + 1;$$

En la mayoría de los hardware esto no será ejecutado en una operación **indivisible** (atómica), sino que será implementada en tres instrucciones distintas:

- (1) cargar el valor de  $X$  en algún registro (o en la cima del stack);
- (2) incrementar en uno el valor en el registro; y



Aunque las variables compartidas aparecen como una forma directa de pasar información entre procesos, su uso sin restricción no es confiable y es inseguro debido a problemas de actualizaciones múltiples.

Considere dos procesos actualizando una variable compartida  $X$ , con la asignación

$$X = X + 1;$$

En la mayoría de los hardware esto no será ejecutado en una operación **indivisible** (atómica), sino que será implementada en tres instrucciones distintas:

- (1) cargar el valor de  $X$  en algún registro (o en la cima del stack);
- (2) incrementar en uno el valor en el registro; y
- (3) almacenar el valor en el registro de regreso a  $X$ .

Como las tres operaciones no son indivisibles, dos procesos actualizando la variable simultáneamente podrían entrelazar sus acciones y producir un resultado incorrecto. Por ejemplo, si  $X$  era originalmente 5, los dos procesos podrían cargar 5 en sus registros, incrementar y entonces almacenar 6.

Como las tres operaciones no son indivisibles, dos procesos actualizando la variable simultáneamente podrían entrelazar sus acciones y producir un resultado incorrecto. Por ejemplo, si  $X$  era originalmente 5, los dos procesos podrían cargar 5 en sus registros, incrementar y entonces almacenar 6.

## Condiciones de Carrera

Aquellas situaciones en donde dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y exactamente cuándo lo hace, se conocen como **condiciones de carrera**.

Como las tres operaciones no son indivisibles, dos procesos actualizando la variable simultáneamente podrían entrelazar sus acciones y producir un resultado incorrecto. Por ejemplo, si  $X$  era originalmente 5, los dos procesos podrían cargar 5 en sus registros, incrementar y entonces almacenar 6.

## Condiciones de Carrera

Aquellas situaciones en donde dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y exactamente cuándo lo hace, se conocen como **condiciones de carrera**.

Depurar programas que contienen condiciones de carrera no es nada divertido. Los resultados de la mayoría de las ejecuciones de prueba están bien, pero en algún momento poco frecuente ocurrirá algo extraño e inexplicable.

## Ejemplos de Condiciones de Carrera

El siguiente es un ejemplo simple pero común: un spooler de impresión. Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un directorio de spooler especial. Otro proceso, el demonio de impresión, comprueba en forma periódica si hay archivos que deban imprimirse y si los hay, los imprime y luego elimina sus nombres del directorio.

## Ejemplos de Condiciones de Carrera

El siguiente es un ejemplo simple pero común: un spooler de impresión. Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un directorio de spooler especial. Otro proceso, el demonio de impresión, comprueba en forma periódica si hay archivos que deban imprimirse y si los hay, los imprime y luego elimina sus nombres del directorio.

Imagine que nuestro directorio de spooler tiene una cantidad muy grande de ranuras, numeradas como 0, 1, 2, ..., cada una de ellas capaz de contener el nombre de un archivo. Imagine también que hay dos variables compartidas: *sal*, que apunta al siguiente archivo a imprimir, y *ent*, que apunta a la siguiente ranura libre en el directorio. Estas dos variables podrían mantenerse muy bien en un archivo de dos palabras disponible para todos los procesos.

En cierto momento, las ranuras de la 0 a la 3 están vacías (ya se han impreso los archivos) y las ranuras de la 4 a la 6 están llenas (con los nombres de los archivos en la cola de impresión). De manera más o menos simultánea, los procesos A y B desiden que desean poner en cola un archivo para imprimirlo.

---

<sup>1</sup>Si algo puede salir mal, lo hará.

En cierto momento, las ranuras de la 0 a la 3 están vacías (ya se han impreso los archivos) y las ranuras de la 4 a la 6 están llenas (con los nombres de los archivos en la cola de impresión). De manera más o menos simultánea, los procesos A y B desiden que desean poner en cola un archivo para imprimirlo.

En las jurisdicciones en las que se aplica la ley de Murphy<sup>1</sup> podría ocurrir lo siguiente. El proceso A lee *ent* y guarda el valor 7 en una variable local, llamada *siguiente\_ranura\_libre*. Justo entonces ocurre una interrupción de reloj y la CPU decide que el proceso A se ha ejecutado durante un tiempo suficiente, por lo que conmuta al proceso B.

---

<sup>1</sup>Si algo puede salir mal, lo hará.



El proceso B también lee *ent* y también obtiene un 7. De igual forma lo almacena en su variable local *siguiente\_ranura\_libre*. En este instante, ambos procesos piensan que la siguiente ranura libre es la 7.

Ahora el proceso B continúa su ejecución. Almacena el nombre de su archivo en la ranura 7 y actualiza *ent* para que sea 8. Después realiza otras tareas.

El proceso B también lee *ent* y también obtiene un 7. De igual forma lo almacena en su variable local *siguiente\_ranura\_libre*. En este instante, ambos procesos piensan que la siguiente ranura libre es la 7.

Ahora el proceso B continúa su ejecución. Almacena el nombre de su archivo en la ranura 7 y actualiza *ent* para que sea 8. Después realiza otras tareas.

En cierto momento el proceso A se ejecuta de nuevo, partiendo del lugar en el que se quedó. Busca en *siguiente\_ranura\_libre*, encuentra un 7 y escribe el nombre de su archivo en la ranura 7, borrando el nombre que el proceso B acaba de poner ahí. Luego calcula  $\text{siguiente\_ranura\_libre} + 1$ , que es 8 y fija *ent* para que sea 8. El directorio de spooler es ahora internamente consistente, por lo que el demonio de impresión no detectará nada incorrecto, pero el proceso B nunca recibirá ninguna salida.

## ¿Qué ocasiona las condiciones de carrera?

Las dificultades mencionadas en los ejemplos anteriores ocurrieron debido a que el proceso B empezó a utilizar una de las variables compartidas antes de que el proceso A terminara con ella.

## ¿Qué ocasiona las condiciones de carrera?

Las dificultades mencionadas en los ejemplos anteriores ocurrieron debido a que el proceso B empezó a utilizar una de las variables compartidas antes de que el proceso A terminara con ella.

## ¿Cómo evitar las condiciones de carrera?

El problema de evitar las condiciones de carrera también se puede formular de una manera abstracta. Parte del tiempo, un proceso está ocupado realizando cálculos internos y otras cosas que no producen condiciones de carrera. Sin embargo, algunas veces un proceso tiene que acceder a la memoria compartida o a archivos compartidos, o hacer otras cosas críticas que pueden producir carreras.

## ¿Qué ocasiona las condiciones de carrera?

Las dificultades mencionadas en los ejemplos anteriores ocurrieron debido a que el proceso B empezó a utilizar una de las variables compartidas antes de que el proceso A terminara con ella.

## ¿Cómo evitar las condiciones de carrera?

El problema de evitar las condiciones de carrera también se puede formular de una manera abstracta. Parte del tiempo, un proceso está ocupado realizando cálculos internos y otras cosas que no producen condiciones de carrera. Sin embargo, algunas veces un proceso tiene que acceder a la memoria compartida o a archivos compartidos, o hacer otras cosas críticas que pueden producir carreras.

Esa parte del programa en la que se accede a la memoria compartida se conoce como **región crítica** o **sección crítica**.

## ¿Cómo evitar las condiciones de carrera? Cont...

Si pudieramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las carreras.

## ¿Cómo evitar las condiciones de carrera? Cont...

Si pudieramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las carreras.

Aunque este requerimiento evita las condiciones de carrera, ([5]) no es suficiente para que los procesos en paralelo cooperen de la manera correcta y eficiente al utilizar datos compartidos.

Necesitamos cumplir con cuatro condiciones para tener una buena solución:

## ¿Cómo evitar las condiciones de carrera? Cont...

Si pudieramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las carreras.

Aunque este requerimiento evita las condiciones de carrera, ([5]) no es suficiente para que los procesos en paralelo cooperen de la manera correcta y eficiente al utilizar datos compartidos.

Necesitamos cumplir con cuatro condiciones para tener una buena solución:

1. No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.



## ¿Cómo evitar las condiciones de carrera? Cont...

Si pudiéramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las carreras.

Aunque este requerimiento evita las condiciones de carrera, ([5]) no es suficiente para que los procesos en paralelo cooperen de la manera correcta y eficiente al utilizar datos compartidos.

Necesitamos cumplir con cuatro condiciones para tener una buena solución:

1. No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.
2. No pueden hacerse suposiciones acerca de las velocidades o el número de CPUs.

## ¿Cómo evitar las condiciones de carrera? Cont...

Si pudiéramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las carreras.

Aunque este requerimiento evita las condiciones de carrera, ([5]) no es suficiente para que los procesos en paralelo cooperen de la manera correcta y eficiente al utilizar datos compartidos.

Necesitamos cumplir con cuatro condiciones para tener una buena solución:

1. No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.
2. No pueden hacerse suposiciones acerca de las velocidades o el número de CPUs.
3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.

## ¿Cómo evitar las condiciones de carrera? Cont...

Si pudiéramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las carreras.

Aunque este requerimiento evita las condiciones de carrera, ([5]) no es suficiente para que los procesos en paralelo cooperen de la manera correcta y eficiente al utilizar datos compartidos.

Necesitamos cumplir con cuatro condiciones para tener una buena solución:

1. No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.
2. No pueden hacerse suposiciones acerca de las velocidades o el número de CPUs.
3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.
4. Ningún proceso tiene que esperar para siempre para entrar a su región crítica.

## ¿Cómo evitamos las condiciones de carrera?

La clave para evitar problemas aquí y en muchas otras situaciones en las que se involucran la memoria compartida, los archivos compartidos y todo lo demás compartido es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo.

## ¿Cómo evitamos las condiciones de carrera?

La clave para evitar problemas aquí y en muchas otras situaciones en las que se involucran la memoria compartida, los archivos compartidos y todo lo demás compartido es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo.

Dicho en otras palabras, lo que necesitamos es **exclusión mutua**, cierta forma de asegurar que si un proceso está utilizando una variable o archivo compartido, los demás procesos se excluirán de hacer lo mismo.

## Exclusión mutua con espera ocupada

Existen varias proposiciones para lograr la exclusión mutua, de manera que mientras un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso puede entrar a su región crítica y ocasionar problemas.

## Exclusión mutua con espera ocupada

Existen varias proposiciones para lograr la exclusión mutua, de manera que mientras un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso puede entrar a su región crítica y ocasionar problemas.

## Deshabilitando interrupciones

En un sistema con un solo procesador, la solución más simple es hacer que cada proceso deshabilite todas las interrupciones justo antes de entrar a su región crítica y las rehabilite justo después de salir.

## Exclusión mutua con espera ocupada

Existen varias proposiciones para lograr la exclusión mutua, de manera que mientras un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso puede entrar a su región crítica y ocasionar problemas.

## Deshabilitando interrupciones

En un sistema con un solo procesador, la solución más simple es hacer que cada proceso deshabilite todas las interrupciones justo antes de entrar a su región crítica y las rehabilite justo después de salir.

Con las interrupciones deshabilitadas, no pueden ocurrir interrupciones de reloj. Como la CPU solo se conmuta de un proceso a otro como resultado de una interrupción del reloj o de otro tipo, y con las interrupciones desactivadas la CPU no se conmutará a otro proceso. Si un proceso ha deshabilitado las interrupciones, puede examinar y actualizar la memoria compartida sin temor de que algún otro proceso intervenga.



En sistemas operativos el método de deshabilitar interrupciones es poco atractivo, ya que no es conveniente dar a los procesos de usuario el poder de desactivar las interrupciones. Suponga que uno de ellos lo hiciera y nunca las volviera a activar. Ese podría ser el fin del sistema.

En sistemas operativos el método de deshabilitar interrupciones es poco atractivo, ya que no es conveniente dar a los procesos de usuario el poder de desactivar las interrupciones. Suponga que uno de ellos lo hiciera y nunca las volviera a activar. Ese podría ser el fin del sistema.

Además, si el sistema es multiprocesador, al desactivar las interrupciones solo se ve afectada la CPU que ejecutó la instrucción disable. Las demás continuarán ejecutándose y pueden acceder a la memoria compartida.

Por otro lado, con frecuencia es conveniente para el mismo kernel deshabilitar las interrupciones por unas cuantas instrucciones mientras actualiza variables o listas. Por ejemplo, si ocurriera una interrupción mientras la lista de procesos se encuentra en un estado inconsistente, podrían producirse condiciones de carrera.

Por otro lado, con frecuencia es conveniente para el mismo kernel deshabilitar las interrupciones por unas cuantas instrucciones mientras actualiza variables o listas. Por ejemplo, si ocurriera una interrupción mientras la lista de procesos se encuentra en un estado inconsistente, podrían producirse condiciones de carrera.

La conclusión es que deshabilitar interrupciones es a veces útil dentro del mismo sistema operativo, pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario. Sin embargo

## Deshabilitar interrupciones para evitar carreras ya no es solución en Linux

En los primeros núcleos de Linux, había relativamente pocas fuentes de concurrencia. Los sistemas multiproceso simétrico no eran compatibles con el núcleo, y la única causa de ejecución concurrente era la gestión de interrupciones de hardware. Aquel entorno ofrecía sencillez, pero ya no sirve en un mundo que valora el rendimiento en sistemas con cada vez más procesadores, y que insiste en que el sistema responda con rapidez.

## Deshabilitar interrupciones para evitar carreras ya no es solución en Linux

En los primeros núcleos de Linux, había relativamente pocas fuentes de concurrencia. Los sistemas multiproceso simétrico no eran compatibles con el núcleo, y la única causa de ejecución concurrente era la gestión de interrupciones de hardware. Aquel entorno ofrecía sencillez, pero ya no sirve en un mundo que valora el rendimiento en sistemas con cada vez más procesadores, y que insiste en que el sistema responda con rapidez.

La posibilidad de lograr la exclusión mutua al deshabilitar las interrupciones (incluso dentro del kernel) está disminuyendo día con día debido al creciente número de chips multinúcleo que se encuentran hasta en las PCs de bajo rendimiento.

## Variables de candado

Buscando una solución de software. Considere tener una sola variable compartida (de candado), que al principio es 0. Cuando un proceso desea entrar a su región crítica primero evalúa el candado. Si este candado es 0, el proceso lo fija en 1 y entra a la región crítica. Si el candado ya es 1 solo espera hasta que el candado se haga 0.

## Variables de candado

Buscando una solución de software. Considere tener una sola variable compartida (de candado), que al principio es 0. Cuando un proceso desea entrar a su región crítica primero evalúa el candado. Si este candado es 0, el proceso lo fija en 1 y entra a la región crítica. Si el candado ya es 1 solo espera hasta que el candado se haga 0.

Por desgracia, esta idea contiene exactamente el mismo error fatal que vimos en el ejemplo del directorio de spooler. Suponga que un proceso lee el candado y ve que es 0. Antes de que pueda fijar el candado a 1, otro proceso se planifica para ejecutarse y fija el candado a 1. Cuando el primer proceso se ejecuta de nuevo, también fija el candado a 1 y por lo tanto dos procesos podrían encontrarse en sus regiones críticas al mismo tiempo.



## Alternancia estricta

Tercera propuesta de solución:

(a) Proceso 0

```
while(TRUE){  
    while(turno != 0) /* ciclo */;  
    region_critica();  
    turno = 1;  
    region_nocritica();  
}
```

(b) Proceso 1

```
while(TRUE){  
    while(turno != 1) /* ciclo */;  
    region_critica();  
    turno = 0;  
    region_nocritica();  
}
```

## Alternancia estricta

Tercera propuesta de solución:

(a) Proceso 0

```
while(TRUE){  
    while(turno != 0) /* ciclo */;  
    region_critica();  
    turno = 1;  
    region_nocritica();  
}
```

(b) Proceso 1

```
while(TRUE){  
    while(turno != 1) /* ciclo */;  
    region_critica();  
    turno = 0;  
    region_nocritica();  
}
```

En la figura anterior, la variable entera turno (que al principio es 0) da cuenta de acerca de a qué proceso le toca entrar a su región crítica y examinar o actualizar la memoria compartida. Al principio el proceso 0 inspecciona turno, descubre que es 0 y entra a su región crítica. El proceso 1 también descubre que es 0 y por lo tanto se queda en un ciclo estrecho, evaluando turno en forma continua para ver cuando se convierte en 1.

## Espera ocupada

A la acción de evaluar en forma continua una variable hasta que aparezca cierto valor se le conoce como **espera ocupada (busy waiting)**. Se utiliza cuando hay una expectativa razonable de que la espera será corta. A un candado que utiliza la espera ocupada se le conoce como **candado de giro (spinlock)**.

## Espera ocupada

A la acción de evaluar en forma continua una variable hasta que aparezca cierto valor se le conoce como **espera ocupada (busy waiting)**. Se utiliza cuando hay una expectativa razonable de que la espera será corta. A un candado que utiliza la espera ocupada se le conoce como **candado de giro (spinlock)**.

Cuando el proceso 0 sale de la región crítica establece turno a 1, para permitir que el proceso 1 entre a su región crítica. Suponga que el proceso 1 sale rápidamente de su región crítica, de manera que ambos procesos se encuentran en sus regiones no críticas, con turno establecido en 0.

## Espera ocupada

A la acción de evaluar en forma continua una variable hasta que aparezca cierto valor se le conoce como **espera ocupada (busy waiting)**. Se utiliza cuando hay una expectativa razonable de que la espera será corta. A un candado que utiliza la espera ocupada se le conoce como **candado de giro (spinlock)**.

Cuando el proceso 0 sale de la región crítica establece turno a 1, para permitir que el proceso 1 entre a su región crítica. Suponga que el proceso 1 sale rápidamente de su región crítica, de manera que ambos procesos se encuentran en sus regiones no críticas, con turno establecido en 0.

Si ahora el proceso 0 ejecuta todo su ciclo rápidamente, saliendo de su región crítica y estableciendo turno a 1. En este punto, turno es 1, y ambos procesos se están ejecutando en sus regiones no críticas.

De repente, el proceso 0 termina su región no crítica y regresa a la parte superior de su ciclo. por desgracia no puede entrar a su región crítica ahora, debido a que turno es 1 y el proceso 1 está ocupado con su región no crítica. El proceso 0 espera en su ciclo while hasta que el proceso 1 establezca turno a 0. Dicho de otra forma, tomar turnos no es una buena idea cuando uno de los procesos es mucho más lento que el otro.

De repente, el proceso 0 termina su región no crítica y regresa a la parte superior de su ciclo. por desgracia no puede entrar a su región crítica ahora, debido a que turno es 1 y el proceso 1 está ocupado con su región no crítica. El proceso 0 espera en su ciclo while hasta que el proceso 1 establezca turno a 0. Dicho de otra forma, tomar turnos no es una buena idea cuando uno de los procesos es mucho más lento que el otro.

Esta situación viola la condición 3 antes establecida: el proceso 0 está siendo bloqueado por un proceso que no está en su región crítica.

De repente, el proceso 0 termina su región no crítica y regresa a la parte superior de su ciclo. por desgracia no puede entrar a su región crítica ahora, debido a que turno es 1 y el proceso 1 está ocupado con su región no crítica. El proceso 0 espera en su ciclo while hasta que el proceso 1 establezca turno a 0. Dicho de otra forma, tomar turnos no es una buena idea cuando uno de los procesos es mucho más lento que el otro.

Esta situación viola la condición 3 antes establecida: el proceso 0 está siendo bloqueado por un proceso que no está en su región crítica.

De hecho, esta solución requiere que los dos procesos se alternen de manera estricta al entrar en sus regiones críticas. Aunque este algoritmo evita todas las condiciones de carrera, en realidad no es un candidato serio como solución, ya que viola la condición 3.



En resumen, hasta ahora tenemos:

En resumen, hasta ahora tenemos:

## Sección crítica

Una secuencia de instrucciones que debe parecer que se ejecuta indivisiblemente es llamada una **sección crítica**.

En resumen, hasta ahora tenemos:

## Sección crítica

Una secuencia de instrucciones que debe parecer que se ejecuta indivisiblemente es llamada una **sección crítica**.

## Exclusión mutua

La sincronización requerida para proteger una sección crítica es conocida como **exclusión mutua**.

En resumen, hasta ahora tenemos:

## Sección crítica

Una secuencia de instrucciones que debe parecer que se ejecuta indivisiblemente es llamada una **sección crítica**.

## Exclusión mútua

La sincronización requerida para proteger una sección crítica es conocida como **exclusión mutua**.

## Comentario

La atomicidad, aunque ausente de la operación de asignación, se asume que está presente en el nivel de memoria. Entonces, si un proceso está ejecutando  $X = 5$ , simultáneamente con otro ejecutando  $X = 6$ , el resultado será 5 o 6 (no algún otro valor). Si esto no fuera cierto, sería difícil razonar acerca de programas concurrentes o implementar niveles de atomicidad mayores tales como la sincronización de exclusión mutua.

## Algoritmo de Peterson

Al combinar la idea de tomar turnos con la idea de las variables de candado y las variables de advertencia, un matemático holandés llamado T. Dekker fue el primero en idear una solución de software para el problema de la exclusión mutua que no requiere de una alternancia estricta. Un análisis del algoritmo de Dekker se puede consultar en [2].

## Algoritmo de Peterson

Al combinar la idea de tomar turnos con la idea de las variables de candado y las variables de advertencia, un matemático holandés llamado T. Dekker fue el primero en idear una solución de software para el problema de la exclusión mutua que no requiere de una alternancia estricta. Un análisis del algoritmo de Dekker se puede consultar en [2].

En 1981, G. L. Peterson descubrió una manera mucho más simple de lograr la exclusión mutua, con lo cual la solución de Dekker se hizo obsoleta. el algoritmo de Peterson se muestra en la siguiente página. Este algoritmo consiste de dos procedimientos escritos en ANSI C, lo cual significa que se deben suministrar prototipos para todas las funciones definidas y utilizadas. Para ahorrar espacio aquí se omiten.

## Algortimo de Peterson

```
#define FALSE 0  
#define TRUE 1  
#define N 2
```

```
int turno;  
int interesado[N];
```

```
void entrar_region(int proceso){  
    int otro = 1 - proceso;  
    interesado[proceso] = TRUE;  
    turno = proceso;  
    while(turno == proceso && interesado[otro] == TRUE);  
}
```

```
void salir_region(int proceso){  
    interesado[proceso] = FALSE;  
}
```

## Algoritmo de Peterson

Antes de utilizar las variables compartidas (es decir, antes de entrar a su región crítica), cada proceso llama a *entrar\_region* con su número de proceso (0 o 1) como parámetro. Este llamado hará que espere, si es necesario, hasta que sea seguro entrar.



## Algoritmo de Peterson

Antes de utilizar las variables compartidas (es decir, antes de entrar a su región crítica), cada proceso llama a *entrar\_region* con su número de proceso (0 o 1) como parámetro. Este llamado hará que espere, si es necesario, hasta que sea seguro entrar.

Una vez que haya terminado con las variables compartidas, el proceso llama a *salir\_region* para indicar que ha terminado y permitir que algún otro proceso entre si así lo desea.

## Algoritmo de Peterson

Antes de utilizar las variables compartidas (es decir, antes de entrar a su región crítica), cada proceso llama a *entrar\_region* con su número de proceso (0 o 1) como parámetro. Este llamado hará que espere, si es necesario, hasta que sea seguro entrar.

Una vez que haya terminado con las variables compartidas, el proceso llama a *salir\_region* para indicar que ha terminado y permitir que algún otro proceso entre si así lo desea.

### Véamos cómo funciona esta solución

Al principio ningún proceso se encuentra en su región crítica. Ahora el proceso 0 llama a *entrar\_region*. Indica su interés estableciendo su elemento del arreglo y fija *turno* a cero.

Como el proceso 1 no está interesado, *entrar\_region* regresa de inmediato. Si ahora el proceso 1 hace una llamada a *entrar\_region*, se quedará ahí hasta que *interesado[0]* sea *FALSE*, un evento que solo ocurre cuando el proceso 0 llama a *salir\_region* para salir de la región crítica.

Como el proceso 1 no está interesado, *entrar\_region* regresa de inmediato. Si ahora el proceso 1 hace una llamada a *entrar\_region*, se quedará ahí hasta que *interesado[0]* sea *FALSE*, un evento que solo ocurre cuando el proceso 0 llama a *salir\_region* para salir de la región crítica.

Ahora considere el caso en el que ambos procesos llaman a *entrar\_region* en forma simultanea. Ambos almacenarán su número de proceso (0 o 1) en *turno*. Cualquier almacenamiento que se haya realizado al último es el que cuenta; el primero se sobrescribe y se pierde. Suponga que el proceso 1 almacena al último, por lo que *turno* es 1. Cuando ambos procesos llegan a la instrucción *while*, el proceso 0 la ejecuta 0 veces y entra a su región crítica. El proceso 1 itera y no entra a su región crítica sino hasta que el proceso 0 sale de su región crítica.

La exclusión mutua no es la única sincronización de importancia, si dos procesos no comparten variables entonces no se necesita exclusión mutua.

La exclusión mutua no es la única sincronización de importancia, si dos procesos no comparten variables entonces no se necesita exclusión mutua.

## Sincronización de condición

La sincronización de condición es otro requerimiento significativo y se necesita cuando un proceso desea realizar una operación que puede tener sentido o ser segura solamente si otro proceso ya ha realizado alguna acción o está en algún estado definido.

La exclusión mutua no es la única sincronización de importancia, si dos procesos no comparten variables entonces no se necesita exclusión mutua.

## Sincronización de condición

La sincronización de condición es otro requerimiento significativo y se necesita cuando un proceso desea realizar una operación que puede tener sentido o ser segura solamente si otro proceso ya ha realizado alguna acción o está en algún estado definido.

## Ejemplo

Un ejemplo de sincronización de condición se tiene con el uso de buffers. Dos procesos que intercambian datos podrían funcionar mejor si la comunicación no es directa sino a través de un buffer. Esto tiene la ventaja de que desacopla los procesos y permite pequeñas fluctuaciones en la velocidad a la cual los dos procesos están trabajando.

Por ejemplo, un proceso de entrada podría recibir datos en ráfagas, que deben ser puestas en buffers para el proceso de usuario apropiado.



Por ejemplo, un proceso de entrada podría recibir datos en ráfagas, que deben ser puestas en buffers para el proceso de usuario apropiado.

El uso de un buffer para enlazar dos procesos es común en programas concurrentes y es conocido como un sistema productor-consumidor.






Burns A., Welling A. (2003), Sistemas de Tiempo Real y Lenguajes de Programación (3ª Edición), USA: Addison-Wesley.










Burns A., Welling A. (2003), Sistemas de Tiempo Real y Lenguajes de Programación (3ª Edición), USA: Addison-Wesley.



Dijkstra, E. (1965), Cooperating sequential processes, in F. Genuys (ed.), Programming Languages, Academic Press, London.

-  Burns A., Welling A. (2003), Sistemas de Tiempo Real y Lenguajes de Programación (3ª Edición), USA: Addison-Wesley.
-  Dijkstra, E. (1965), Cooperating sequential processes, in F. Genuys (ed.), Programming Languages, Academic Press, London.
-  Marquez F. (2004), UNIX Programación Avanzada (3ª Edición), España: AlfaOmega.

-  Burns A., Welling A. (2003), Sistemas de Tiempo Real y Lenguajes de Programación (3ª Edición), USA: Addison-Wesley.
-  Dijkstra, E. (1965), Cooperating sequential processes, in F. Genuys (ed.), Programming Languages, Academic Press, London.
-  Marquez F. (2004), UNIX Programación Avanzada (3ª Edición), España: AlfaOmega.
-  Wall, K. et al (2001), Programación en Linux (2ª Edición) al Descubierto, España: Prentice-Hall.

-  Burns A., Welling A. (2003), Sistemas de Tiempo Real y Lenguajes de Programación (3ª Edición), USA: Addison-Wesley.
-  Dijkstra, E. (1965), Cooperating sequential processes, in F. Genuys (ed.), Programming Languages, Academic Press, London.
-  Marquez F. (2004), UNIX Programación Avanzada (3ª Edición), España: AlfaOmega.
-  Wall, K. et al (2001), Programación en Linux (2ª Edición) al Descubierto, España: Prentice-Hall.
-  Tanenbaum A. S., (2009), Sistemas Operativos Modernos (3ª Edición), México: Pearson Educación.