

# SISTEMAS OPERATIVOS I

## UNIDAD II

## PROCESOS Y SUBPROCESOS



INSTITUTO TECNOLÓGICO  
DE MORELIA

Departamento de Sistemas y  
Computación

*Disponible en: [www.benito.org.mx](http://www.benito.org.mx)*

**M.C. Benito Sánchez Raya**

[sanchezraya@hotmail.com](mailto:sanchezraya@hotmail.com)

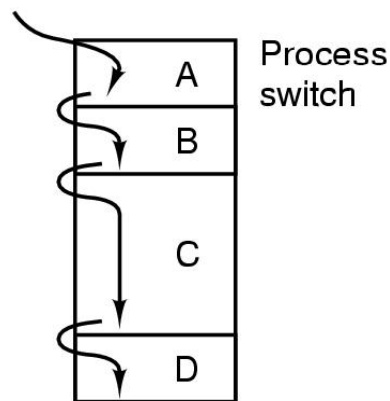
# CONTENIDO

1. Procesos
2. Subprocesos
3. Comunicación entre procesos
4. Problemas clásicos de comunicación entre procesos
5. Calendarización

# 1. PROCESOS

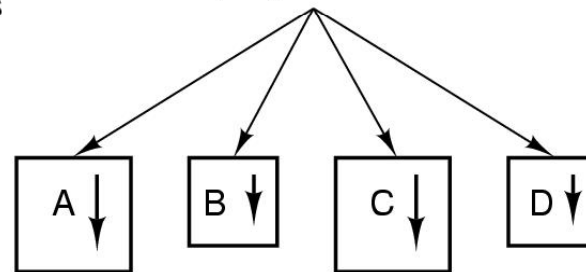
- Pseudoparalelismo
- Paralelismo en sistemas multiprocesador

One program counter

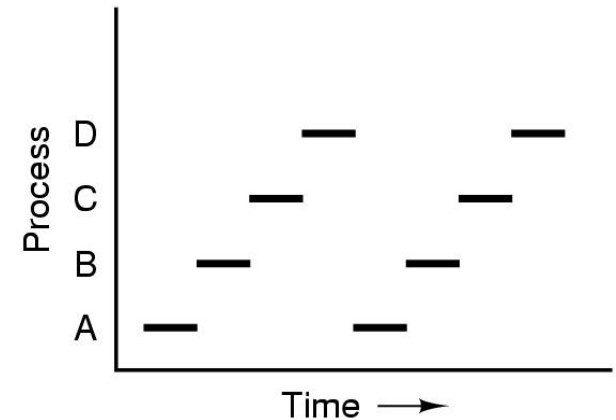


(a)

Four program counters



(b)



(c)

- a) Multiprogramación b) Procesos secuenciales independientes  
c) Sólo un programa activo a la vez

# 1.1. El Modelo de procesos

- Multiprogramación → conmutación entre procesos
- Priorizado de procesos
- Calendarización de procesos


## 1.2. Creación de procesos

- a) Al inicializar el sistema
  - Win → Servicios, Linux → Demonios
  - Ya sea en primer o segundo plano
- b) En llamadas al sistema
- c) Solicitud de usuario para crearlo
  - Comando o aplicación
- d) Inicio de un trabajo por lotes
  - Ejecución de trabajos con la creación de procesos

- Se crean con:
  - Fork + execve
  - CreateProcess
- Padre e hijo tienen distintos espacios de memoria
- Probablemente compartan el código del programa

## 1.3. Terminación de procesos

- a) Terminación normal (voluntaria)
  - Cuando termina su trabajo
    - Exit → Unix, ExitProcess → Win
- b) Terminación por error (voluntaria)
  - Terminación por código
- c) Error fatal (involuntaria)
  - Instrucción no permitida, división por 0, referencia a memoria inexistente, etc

- 
- d) Terminado por otro proceso (involuntaria)
- Kill de otro proceso en Unix, `TerminateProcess` en Win.
  - Deben tener autorización para hacerlo.

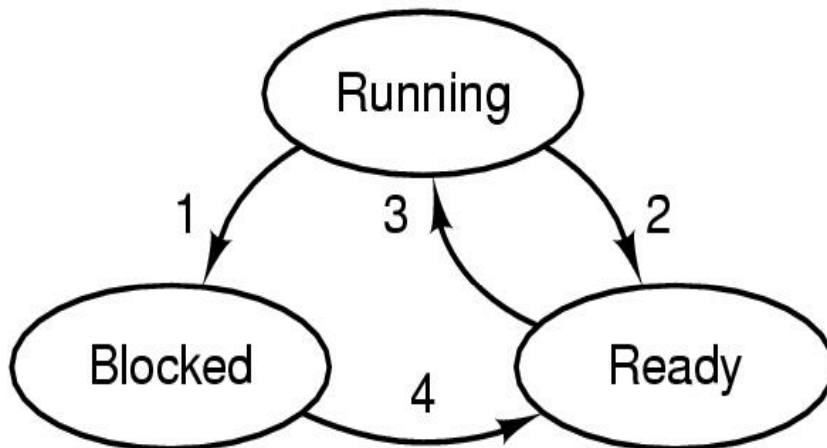


## 1.4. Jerarquía de procesos

- En Win no hay jerarquía de procesos
- En Unix:
  - Se crea una jerarquía de árbol
  - Los procesos padres no pueden “desheredar” a los hijos
  - Ejemplo:
    - Al iniciar init (imagen de arranque) lee un archivo que le indica cuantas terminales hay, generando un proceso nuevo por cada terminal.
    - Los procesos de las terminales esperan a que alguien inicie sesión.
    - Si alguien inicia sesión el shell espera comandos para iniciar procesos por cada comando, y así sucesivamente.

## 1.5. Estados de procesos

- Estados:
  - a) En ejecución → Usando la CPU
  - b) Listo → En espera que se desocupe la CPU
  - c) Bloqueado → En espera de que suceda cierto suceso externo

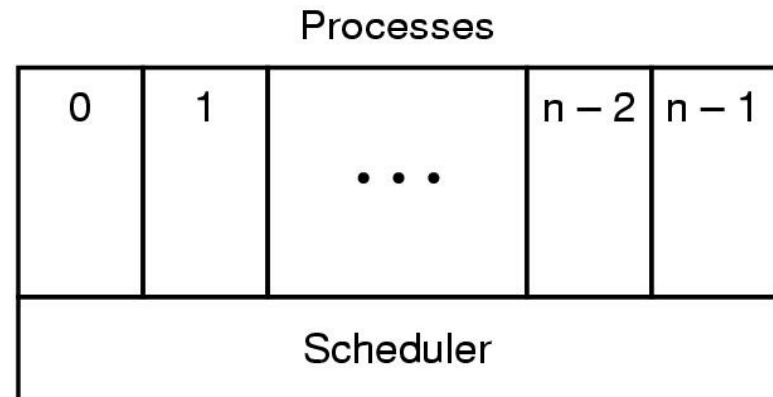


1. El proceso se bloquea para esperar entrada
2. El calendarizador escoge otro proceso
3. El calendarizador escoge éste proceso
4. Ya hay entrada disponible

## Ejemplo:

- El comando *cat file1 file2 file3 | grep finanzas*
  - Se concatenan tres archivos y *grep* selecciona las entradas que contienen la palabra finanzas
  - *Grep* queda bloqueado hasta que *cat* termina.

El calendarizador y los procesos



## 1.6. Implementación de procesos

- El SO mantiene la tabla de procesos, también llamada bloques de control de procesos, la cual contiene:
  - Estado del proceso
  - Su contador del programa
  - Apuntador de pila
  - Asignación de memoria
  - Estado de sus archivos abiertos
  - Información de calendarización

# Campos de una tabla de procesos:

## a) Administración de procesos

- Registros
- Contador de programa
- Palabra del estado del programa
- Apuntador de pila
- Estado del proceso
- Prioridad
- Parametros de calendarización
- ID de proceso
- Proceso padre
- Grupo de procesos
- Señales
- Hora de inicio del proceso
- Tiempo de CPU consumido
- Tiempo de CPU de los hijos
- Hora de la siguiente alarma

# Campos de una tabla de procesos:

- b) Administración de memoria
  - Apuntador a segmento de texto
  - Apuntador a segmento de datos
  - Apuntador a segmento de pila
  
- c) Administración de archivos
  - Directorio raíz
  - Directorio de trabajo
  - Descriptores de archivo
  - ID de usuario
  - ID de grupo

- Vector de interrupción
  - Contiene la dirección del procedimiento de servicio de interrupción de cada dispositivo de E/S.
  - Las interrupciones de vaciado de pilas, guardar registros, lo hace una pequeña rutina en ensamblador

- Cuando ocurre una interrupción
  - El hardware mete el contador de programa en la pila, etc.
  - El hardware carga un nuevo contador de programa tomándolo del vector de interrupción.
  - Un procedimiento en ensamblador guarda registros
  - Un procedimiento en ensamblador crea la nueva pila
  - Se ejecuta el servicio de interrupción en C (lee entradas y las pone en un búfer)
  - El calendarizador decide que programa ejecutara ahora
  - Un procedimiento en C regresa al código ensamblador
  - Un procedimiento en ensamblador arranca el nuevo proceso actual

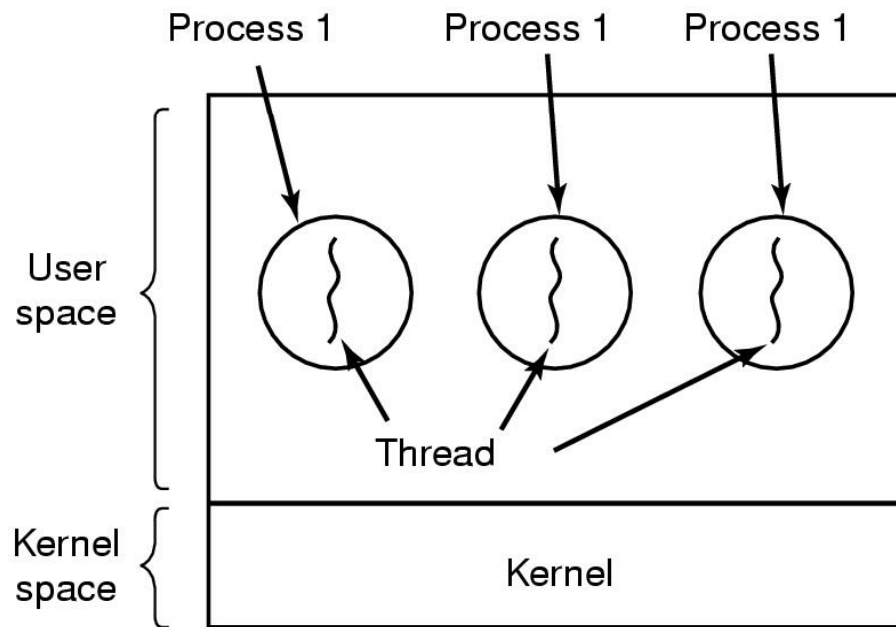


## 2. SUBPROCESOS

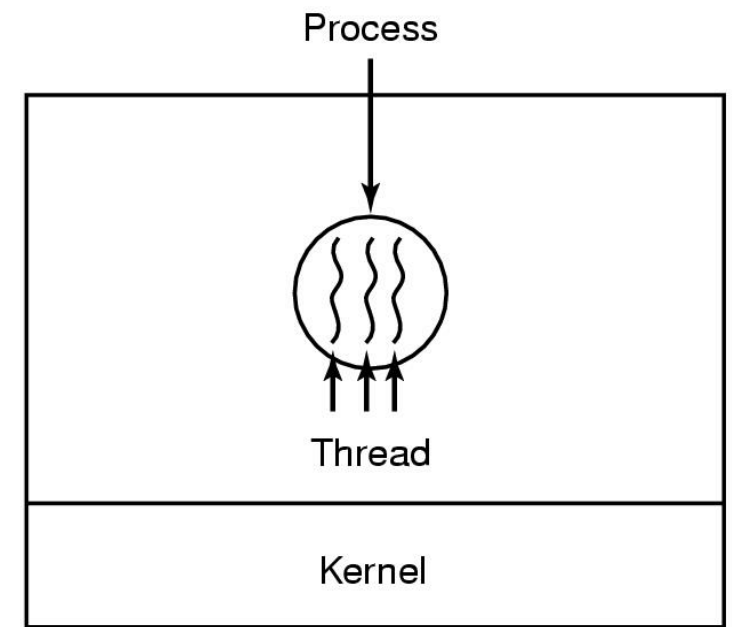
- Antes en los SO un proceso solía tener:
  - Un espacio de direcciones
  - Un sólo subproceso de control.
- Ahora es común tener:
  - Varios subprocesos de control en el mismo espacio de direcciones, operando en forma pseudoparalela como si fueran procesos individuales.

## 2.1. El modelo de subprocesos

- Proceso:
  - Es un agrupamiento de recursos relacionados, para una mejor administración.
    - Espacio de direcciones, variables globales, archivos abiertos, procesos hijos, alarmas pendientes, señales, etc.
- Subproceso:
  - Son las entidades que se calendarizan para ejecutarse en el CPU
    - Tratan de simular “paralelismo”
    - Elementos: contador de programa, registros, pila, etc.



(a)



(b)

a) Tres procesos c/u con un subproceso b) Un proceso con tres subprocesos.

- La protección entre subprocesos es muy difícil llevarla a cabo y no es necesaria.
  - Un proceso crea subprocesos para colaborar, no para dañarse.
  - Los subprocesos se usan para colaborar en una misma tarea en forma “paralela”.



- **Elementos por proceso que comparten sus subprocesos:**

- Espacio de direcciones
- Variables globales
- Archivos abiertos
- Procesos hijos
- Alarmas pendientes
- Señales y manejadores de señales
- Información contable

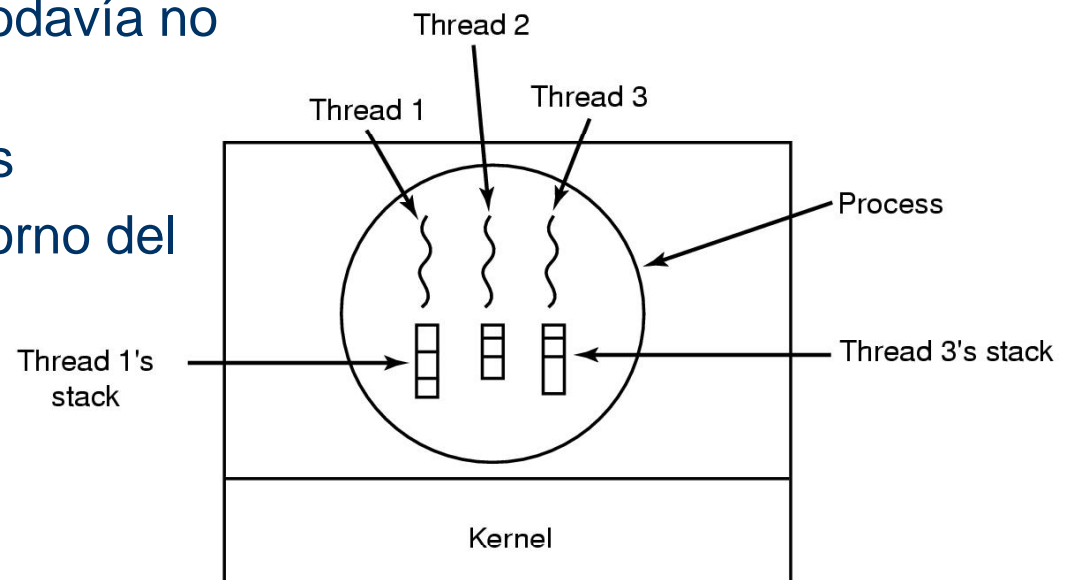


- **Elementos privados de cada subproceso:**

- Contador de programa
- Registros
- Pila
- Estado

- Pila del subproceso:

- Procedimiento en ejecución, que todavía no se retorna
- Variables locales
- Dirección de retorno del procedimiento



**Cada subproceso tiene su propia pila**

- Procedimientos de biblioteca de subprocesos:
  - Thread\_create
  - Thread\_exit
  - Thread\_wait → Bloqueado
  - Thread\_yield → Cede el CPU al siguiente



## 2.2. Uso de subprocesos

- Objetivos:
  - Descomponer una aplicación en múltiples subprocesos secuenciales que se ejecuten casi en paralelo.
  - Son más fáciles de crear y destruir
  - Mayor desempeño al traslapar actividades
  - Indispensables en sistemas multiprocesador
  - En aplicaciones que requieren mucho CPU y que casi no se bloquean, no se recomiendan subprocesos
    - (calcular número primos, juego de ajedrez, etc)

- Ejemplos:

1. Un procesador de textos:

- Corrección ortográfica en segundo plano.
- Paginación en segundo plano
- Respaldo automático cada cierto tiempo
- Lectura del teclado

2. Un antivirus:

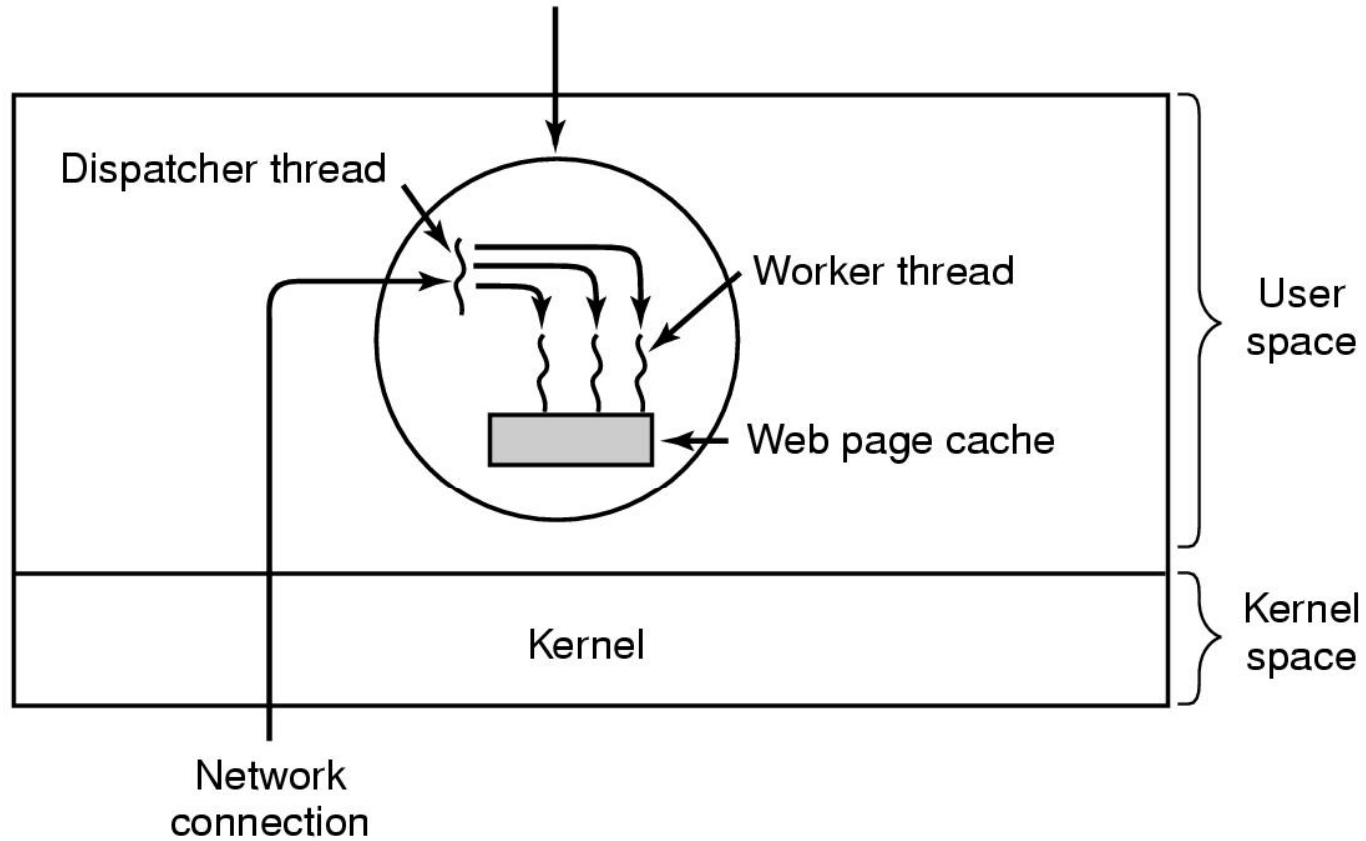
- Centinelas en:
  - E-mail
  - Mensajería
  - Intrusos
  - Unidades extraíbles
  - Actualizaciones (LiveUpdate)
  - Vigencia licencia – contrato

### 3. Servidor Web

- Web Caché
  - Mantiene en la memoria principal las páginas más visitadas del sitio.
- Hay dos tipos de subprocesos
  - a) Despachador
  - b) Trabajador

- Cuando llega una solicitud el subproceso despachador busca un subproceso trabajador desocupado (bloqueado) para mandarle la petición, entrando este a estado listo para ser calendarizado.
- Si el subproceso trabajador encuentra la página solicitada en caché la envía.
- Si no encuentra la página hace una operación *read* a disco y se bloquea, cediendo el CPU al siguiente subproceso listo.

Web server process



```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

Despachador

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page)  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

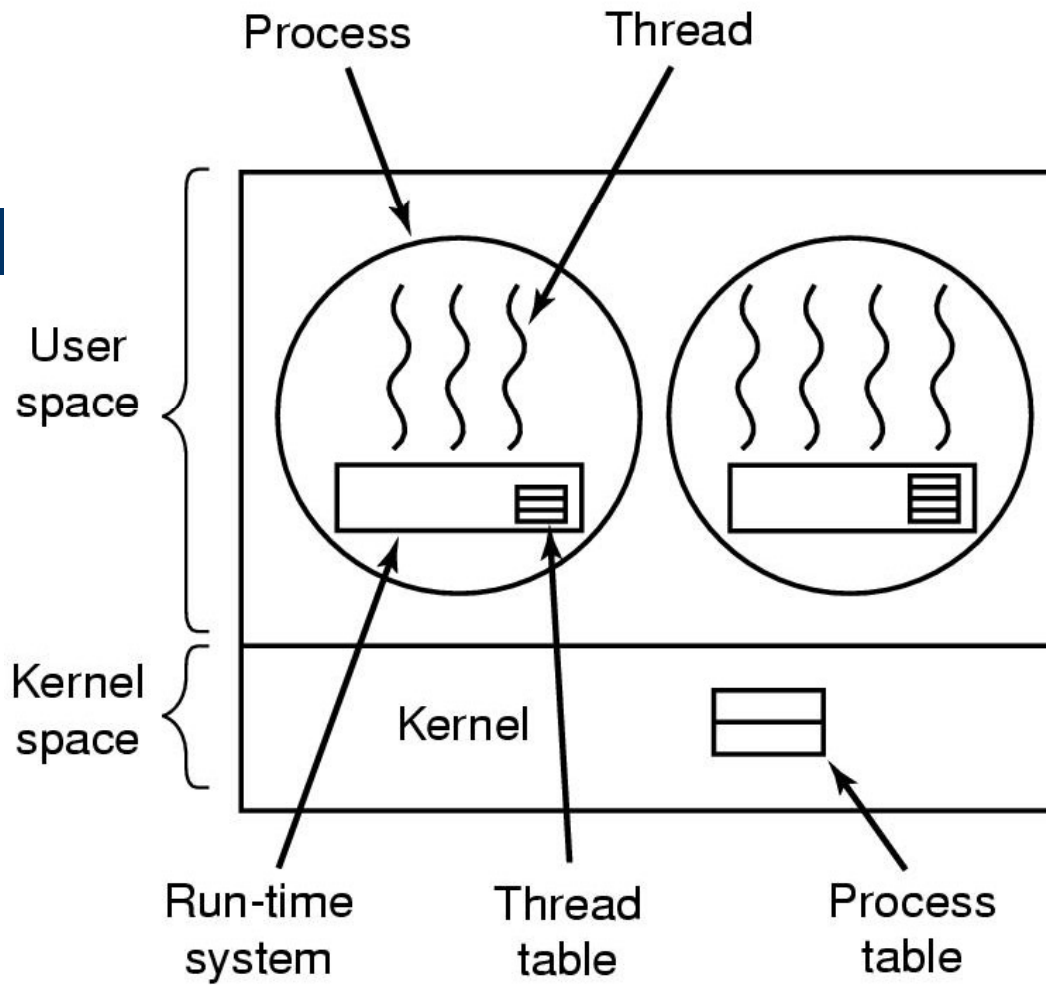
Trabajador

## 2.3. Implementación de subprocesos en espacio de usuario

- Inicialmente así trabajaban los SO
- Los subprocesos se ejecutan en modo de usuario
- Cada proceso requiere su propia tabla de subprocesos
- La conmutación entre subprocesos no requiere llamadas al kernel y suele ser más rápida.
- Tienen mayor desempeño
- Cada proceso puede tener su propio algoritmo de calendarización

- En llamadas bloqueantes podría detener a todos los demás subprocessos.
- Si un subprocesso empieza a ejecutarse, ningún otro subprocesso (del mismo proceso) podrá hacerlo a menos que le cedan el CPU.
- Si un subprocesso hace una llamada al sistema y se bloquea, es probable que bloquee a todo el proceso.

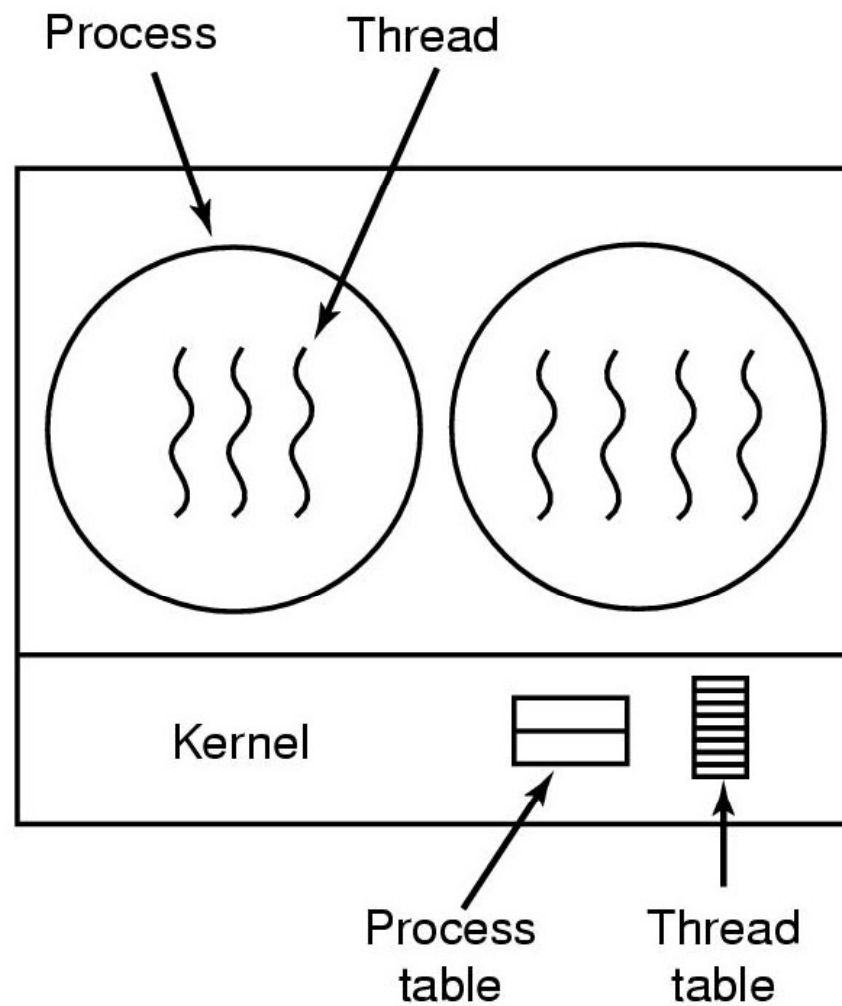




**Sistema de subprocessos en el nivel de usuario**

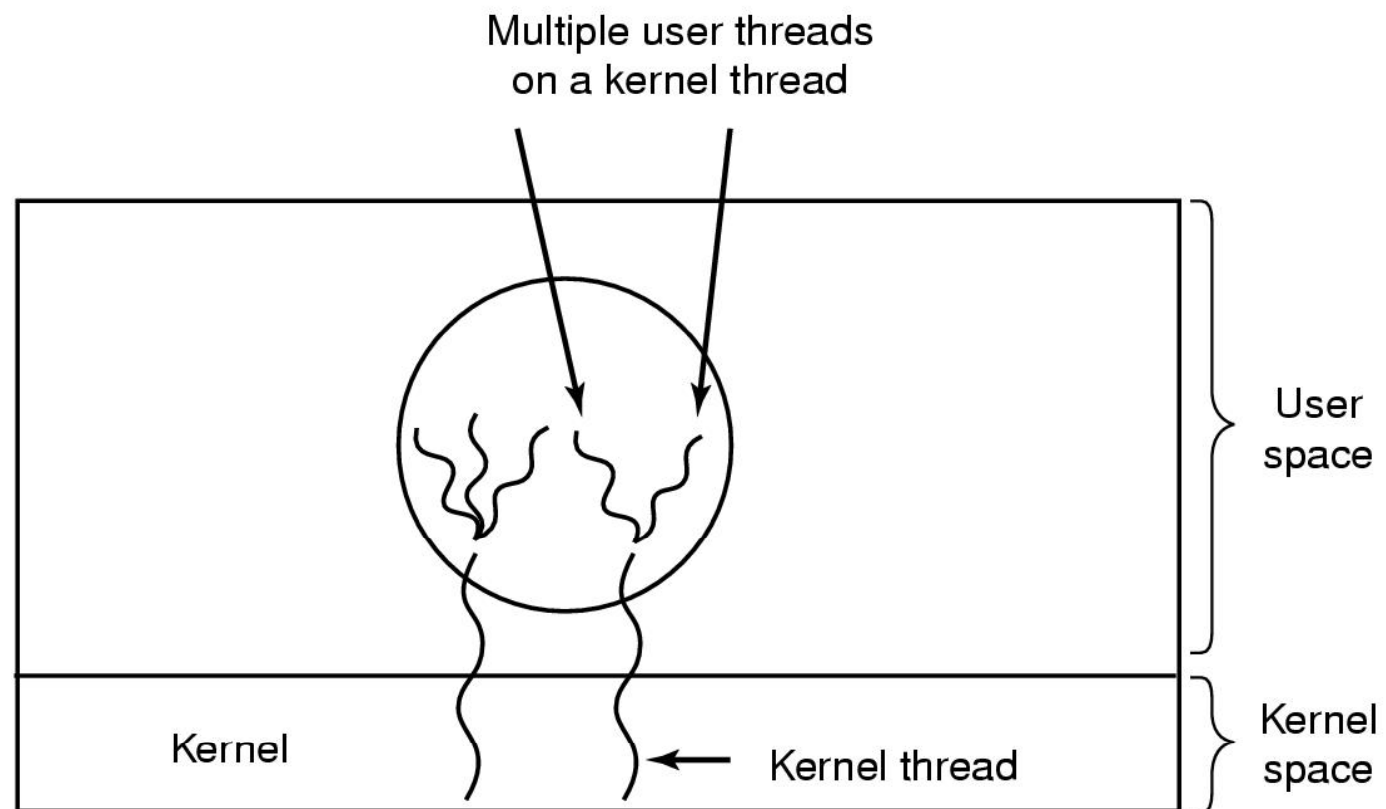
## 2.4. Implementación de subprocesos en el Kernel

- El kernel maneja las tablas de procesos y subprocesos
- Cuando un subproceso se bloquea, el kernel puede ceder el CPU a un subproceso del mismo proceso o de otro proceso.
- Las llamadas al sistemas son más “costosas”
- Es más “costoso” crear y destruir subprocesos en el kernel.
  - Se reciclan o reusan las estructuras de los subprocesos.
  - No se destruyen, solo se marcan como no ejecutables.



Sistema de subprocesos en el kernel

## 2.5. Implementaciones Híbridas

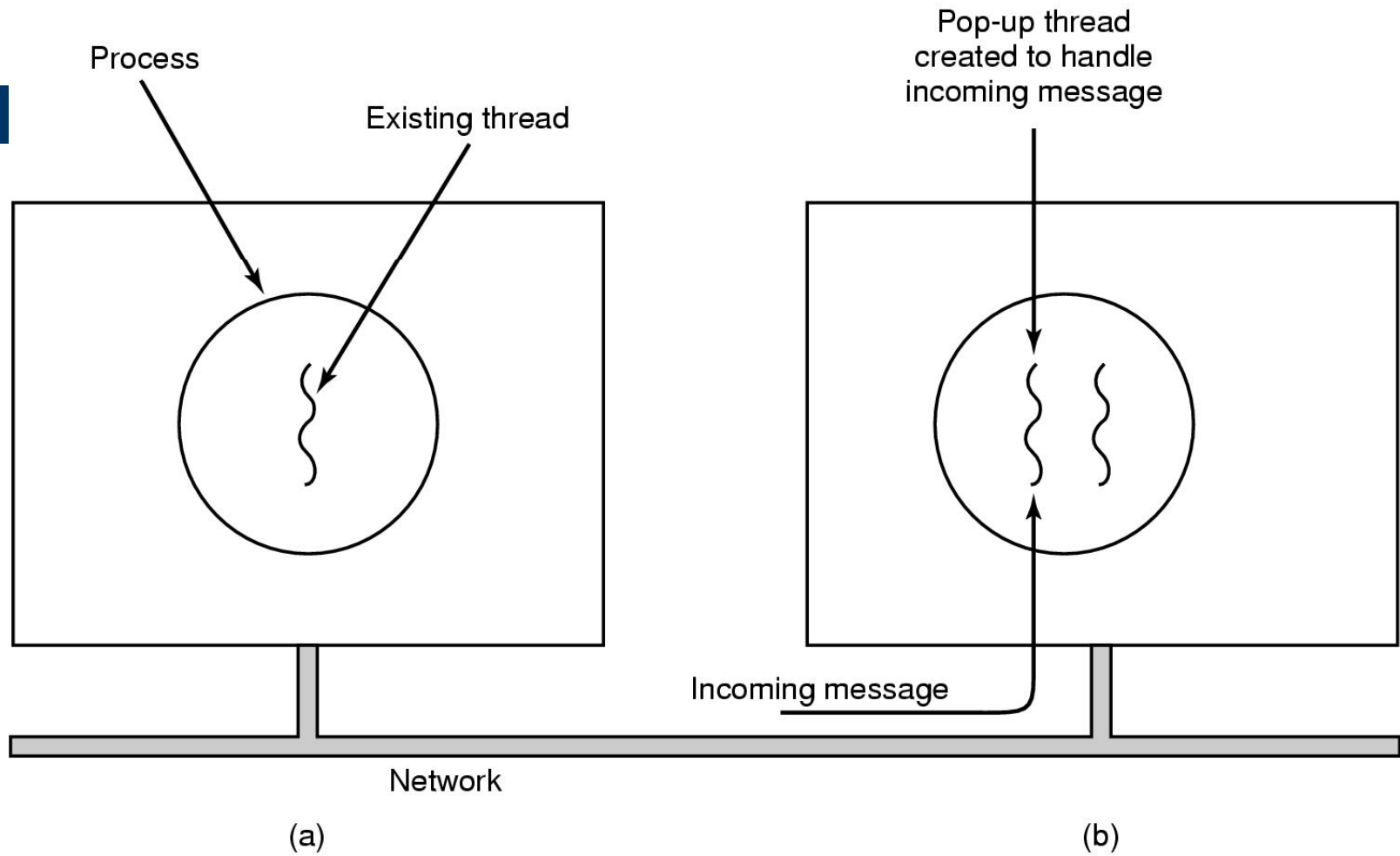


- Es una implementación híbrida de ambas implementaciones.
- Trata de usar la funcionalidad de los procesos kernel pero con el desempeño y flexibilidad de los modo usuario
- Si un subproceso se bloquea con una llamada al sistema, no debiera bloquearse el proceso, sino darle el control a un subproceso del mismo proceso.

## 2.7. Subprocesos emergentes

- Subproceso emergente:
  - Subproceso que se encarga del manejo de mensajes *receive* para que no quede el proceso en modo bloqueante.
- Ventaja de procesos nuevos:
  - No tienen historial
  - Inician desde cero
  - Esto agiliza su creación

- Se generan en cuanto llega un mensaje
- Es mejor que sea creado en el kernel, para mayor facilidad de acceso a los dispositivos de E/S. Aunque podría causar más daño en kernel que en modo usuario.
- Ejemplos:
  - Email, virus, intrusos, etc.



Creación de un subproceso cuando llega un mensaje. a) Antes b) Después



## 2.8. Convertir código de un proceso a código de múltiples procesos

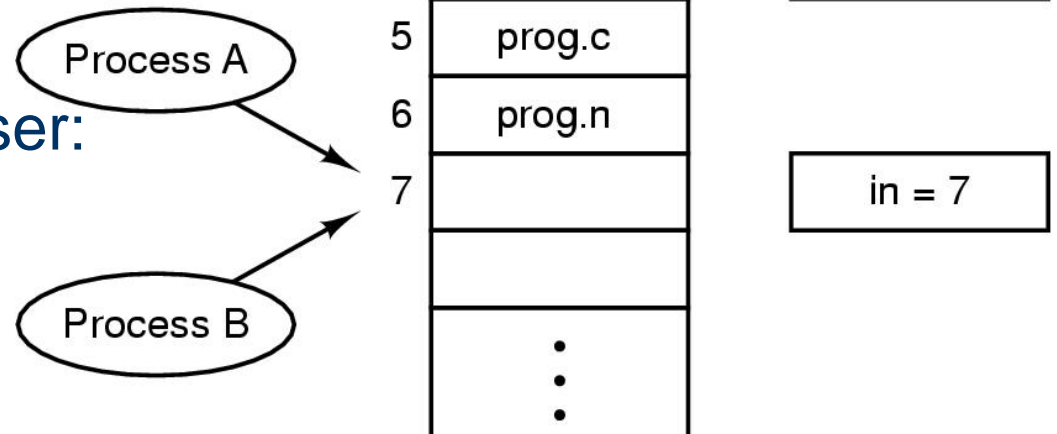
- Detalles de implementación que se deben analizar para programar subprocessos.
  - Revisar la Bibliografía.

## 3. COMUNICACIÓN ENTRE PROCESOS

- Objetivos:
  - Enviar información de un proceso a otro
  - No estorbarse los procesos entre si
  - Ordenamiento correcto cuando un proceso depende de otros
- Aplica a procesos y subprocessos

## 3.1. Condiciones de competencia

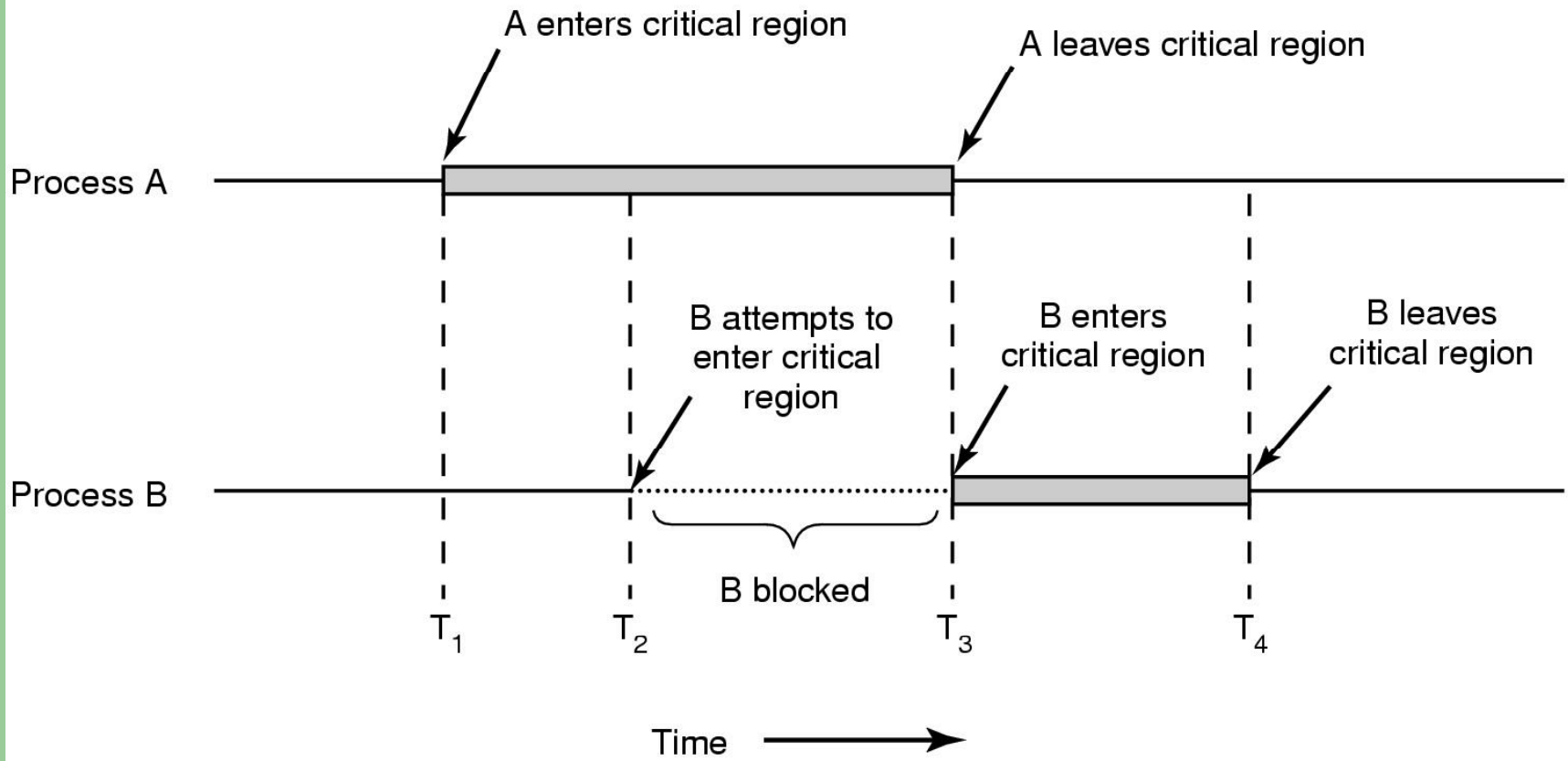
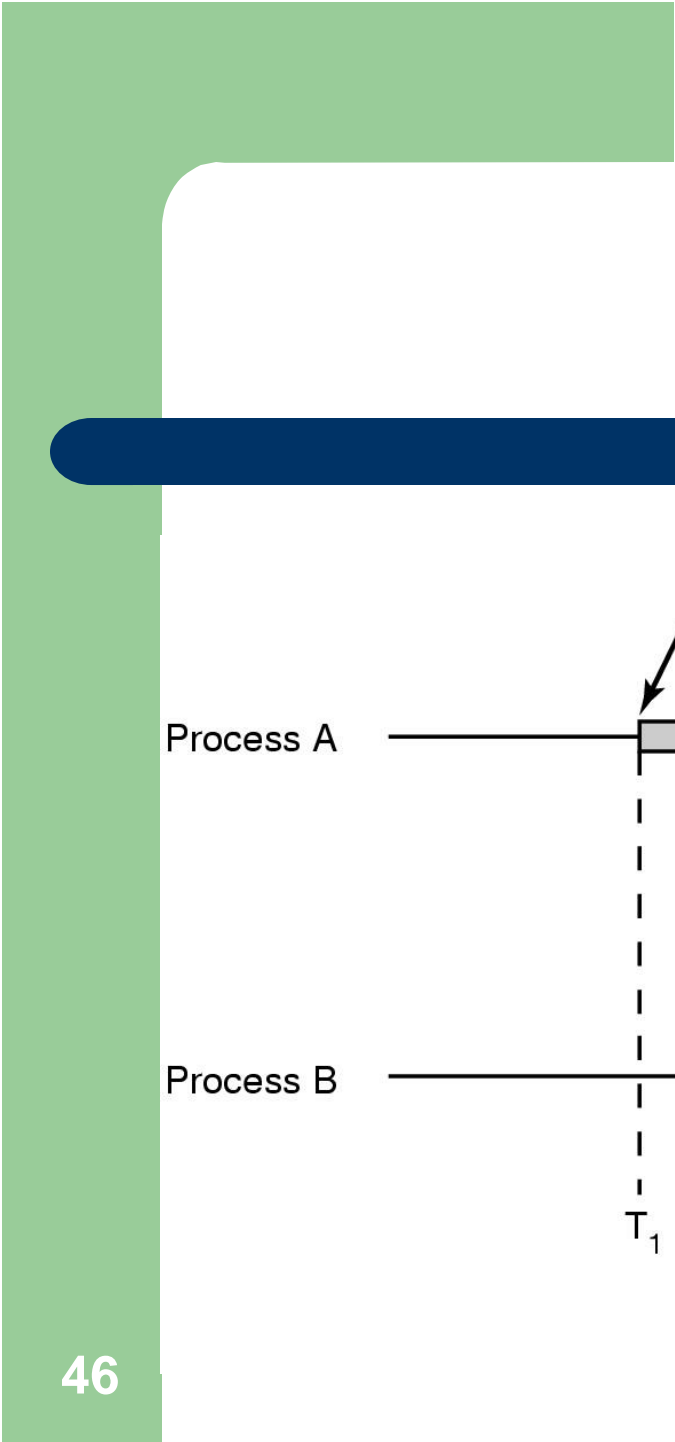
- Se originan cuando se comparten recursos
- La idea es controlar la concurrencia
- El almacenamiento compartido puede ser: memoria principal, archivos, etc.



## 3.2. Regiones críticas

- El objetivo es evitar las condiciones de competencia
- Se requiere exclusión mutua
- Región o sección crítica:
  - Parte del programa que tiene acceso a recursos compartidos (memoria)
- La idea es asegurar que dos o más procesos nunca estén al mismo tiempo en sus regiones críticas.

- Condiciones básicas para evitar condiciones de competencia:
  1. Dos procesos no pueden estar al mismo tiempo dentro de sus regiones críticas.
  2. No pueden hacer suposiciones sobre las velocidades ni el número de CPUs.
  3. Ningún proceso en ejecución fuera de su región crítica puede bloquear a otros.
  4. Ningún proceso deberá tener que esperar de manera indefinida para entrar en su región crítica.



## 3.3. Exclusión mutua con espera activa

- Se analizan diversas propuestas:
  1. Inhabilitación de interrupciones
  2. Variables de bloqueo
  3. Alternancia estricta
  4. Solución de Peterson
  5. La instrucción TSL

Todas las propuestas requieren espera activa

## 1. Inhabilitación de interrupciones

- Un proceso en cuanto entra a su región crítica inhabilita las interrupciones, para hacer los cambios deseados sin intromisiones
- No es recomendable para procesos modo usuario.
  - Por el riesgo de las interrupciones
- Desventaja:
  - Podría ya no habilitar las interrupciones
  - En equipos multiprocesador solo inhabilitaría su CPU y el resto podrían acceder a la memoria compartida.



## 2. Variables de bloqueo

- Uso de una variable global con valores de 0 y 1.
- Si tiene 0:
  - Un proceso la pone en 1 y entra a su región crítica.
  - Al finalizar la pone nuevamente en 0.
- Desventaja:
  - ¿Qué pasa si dos procesos la leen casi al mismo tiempo?

### 3. Alternancia estricta

- Se usa una variable que va cambiando de acuerdo al número de procesos
- Proceso:
  - Se inicia en 0
  - Sólo puede entrar a la región crítica el proceso 0
  - Al finalizar la incrementa a 1, dándole el turno al proceso 1.
  - Este al finalizar la incrementa a 2 y así sucesivamente.
- Desventaja:
  - Puede tocarle el turno a un proceso que no esta en la región crítica.
    - Habría desperdicio en el uso del recurso compartido, y
    - Podría bloquear al resto de los procesos

## 4. Solución de Peterson

- Se emplea el algoritmo de Peterson
- Usa variables globales (compartidas)
  - $N \rightarrow$  Número de procesos
  - Turno  $\rightarrow$  ¿A quien le toca?
  - Interesado  $[N] \rightarrow$  Todos los que quieren entrar. Inicialmente todos en 0 (FALSO).
- Si dos tratan de entrar al mismo tiempo, se sobrescriben los valores
- El while es el que permite entrar a la región crítica.

```

#define FALSE 0
#define TRUE 1
#define N      2          /* number of processes */

int turn;                /* whose turn is it? */
int interested[N];      /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;          /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;      /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}

```

## 5. La instrucción TSL (Test and Set Lock)

- “Probar y establecer bloqueo”
- Es una instrucción en hardware de algunas computadoras multiprocesador
  - TSL RX, BLOQUEO*
- Proceso:
  - BLOQUEO=0 Disponible, BLOQUEO<>0 Bloqueado
  - Lee el contenido de la palabra de memoria BLOQUEO, y lo coloca en el registro RX.
  - Después guarda un valor distinto de cero en la palabra BLOQUEO.
  - TSL cierra el bus de memoria para impedir que otros CPUs accedan a la memoria.
  - Compara el registro

- Cuando BLOQUEO ES 0, cualquier proceso la puede poner en 1 con TSL y hacer uso de la memoria compartida, al terminar la deja nuevamente en 0.
- Con el manejo del registro se garantiza que las operaciones son indivisibles

enter\_region:

```
TSL REGISTER,LOCK           | copy lock to register and set lock to 1
CMP REGISTER,#0             | was lock zero?
JNE enter_region            | if it was non zero, lock was set, so loop
RET | return to caller; critical region entered
```

leave\_region:

```
MOVE LOCK,#0                | store a 0 in lock
RET | return to caller
```

## 3.4. Activar y desactivar

- Los métodos anteriores requerían espera activa
- Estos métodos se bloquean en lugar de esperar de manera activa.

# Productor-Consumidor

- Dos procesos comparten un buffer de tamaño fijo.
- El productor coloca información.
- El consumidor la extrae.
- Pueden ser  $m$  productores y  $n$  consumidores.





- Premisas:

- Cuando el productor quiere colocar información y el buffer esta lleno se debe bloquear. Y desbloquearse hasta que el consumidor extraiga un valor.
- Cuando el consumidor quiere extraer y el buffer esta vacío, se bloquea hasta que el productor coloca algún valor.

- Se requiere una variable global que lleve la cuenta de cuantos elementos hay en el buffer (cuenta).
- Cada proceso determina si el otro debe activarse o no, de ser así lo activa.

```

#define N 100
int count = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

void consumer(void)
{
    int item;


    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}

```

*/\* number of slots in the buffer \*/*  
*/\* number of items in the buffer \*/*

*/\* repeat forever \*/*  
*/\* generate next item \*/*  
*/\* if buffer is full, go to sleep \*/*  
*/\* put item in buffer \*/*  
*/\* increment count of items in buffer \*/*  
*/\* was buffer empty? \*/*

*/\* repeat forever \*/*  
*/\* if buffer is empty, got to sleep \*/*  
*/\* take item out of buffer \*/*  
*/\* decrement count of items in buffer \*/*  
*/\* was buffer full? \*/*  
*/\* print item \*/*



- Inconvenientes:

- Podían quedarse ambos suspendidos:

- Cuando el consumidor lea cuenta y antes de sleep sale de ejecución, cuenta =0.
    - Entonces el productor coloca un valor, cuenta =1, y despierta a consumidor, pero como no esta bloqueado se pierde esta llamada.
    - Al tomar el control el consumidor se bloquea porque se quedó con un valor de 0.
    - Entonces el productor llena el buffer y se bloquea también.
    - El problema fue la señal de activar que se perdió.



- Solución:

- Añadir un bit de espera para activar.

- Cuando se intenta activar un proceso que esta activo, se activa el bit.
- Cuando el proceso quiere desactivarse se apaga el bit y el proceso se debe seguir activo.
- No sirve para más de dos procesos.

## 3.5. Semáforos

- Es una variable con valores de 0 hasta n.
- Soluciona la pérdida del “despertar”.
- Si el semáforo = 0
  - No hay llamadas Wakeup
- Si es un valor mayor de 0
  - Hay llamadas wakeup pendientes de ejecutarse.

Sleep = down

Wakeup = up



## – Down

- Determina:

- Si el valor es mayor de 0, decrementa el semáforo.
- Si es igual a 0, se bloquea (desactiva).

- Verificar valor, modificarlo y desactivarse se realiza en una acción atómica.

## – Up

- Incrementa el valor del semáforo. No bloquea a ningún proceso.

- Si uno o más procesos estaban inactivos, se escoge uno al azar para que pueda continuar.
- Ver ejemplo del Productor-Consumidor con semáforos.



sleep = down  
wakeup = up

Down:

Decrementa semáforo  
0 → se bloquea

Up:

Incrementa semáforo

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */
```

## 3.6. Mutexes

- Mutex = Exclusión mutua
- Es una variable que puede estar sólo en dos estados: Bloqueado y Desbloqueado.
- Sólo se requiere de un bit para representarlo
  - 0 = Desbloqueado, !0 = Bloqueado
- Procedimientos:
  - mutex\_lock
  - Mutex\_unlock

- `mutex_lock`
  - Si ya está bloqueado, el proceso invocador se bloquea
  - Si no está bloqueado, la llamada procede y se accede a la región crítica.
- Todos los SO ofrecen un área de memoria compartida a todos los procesos o pueden usar archivos. Incluso puede ser un área del kernel.



mutex\_lock:

TSL REGISTER,MUTEX

| copy mutex to register and set mutex to 1

CMP REGISTER,#0

| was mutex zero?

JZE ok

| if it was zero, mutex was unlocked, so return

CALL thread\_yield

| mutex is busy; schedule another thread

JMP mutex\_lock

| try again later

ok: RET | return to caller; critical region entered

mutex\_unlock:

MOVE MUTEX,#0

| store a 0 in mutex

RET | return to caller

## 3.7. Monitores

- Evita los detalles de implementación de los semáforos
- Ver caso página 115
- Se propone una primitiva de sincronización a más alto nivel, llamada monitor.
- Monitor:
  - Colección de procedimientos, variables y estructuras de datos que se agrupan en un módulo especial.

- Los procesos pueden invocar a los procedimientos de un monitor, pero no acceden a sus datos ni estructuras.
- La exclusión mutua la controlan manteniendo sólo un proceso activo a la vez en un monitor
- El compilador trata distinto a los procedimientos de monitores que al resto de ellos.
- Cuando un proceso llama a un procedimientos de monitor, sus primeras líneas del procedimiento verifican si hay otro proceso dentro del monitor, de ser así el proceso invocador se bloquea, de lo contrario puede entrar.

**monitor** *example*

**integer** *i*;

**condition** *c*;

**procedure** *producer*( );

.

.

.

**end**;

**procedure** *consumer*( );

.

.

.

**end**;

**end monitor**;

- Basta con convertir las regiones críticas en monitores y el compilador se encarga.
- Se puede combinar con variables de condición:
  - Wait = Bloquear
  - Signal = Habilita a otro proceso
- Para evitar dos procesos en ejecución que habilite signal saldrá del monitor.



```

monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;

```

```

procedure producer;
begin
  while true do
  begin
    item = produce_item;
    ProducerConsumer.insert(item)
  end
end;
procedure consumer;
begin
  while true do
  begin
    item = ProducerConsumer.remove;
    consume_item(item)
  end
end;

```

- Java maneja monitores: `synchronized`.
- C no tiene monitores.
- Los semáforos se programan, los monitores no.

Figura siguiente:

- **Productor consumidor con Java**

```

public class ProducerConsumer {
    static final int N = 100;           // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor
    public static void main(String args[] ) {
        p.start();                      // start the producer thread
        c.start();                      // start the consumer thread
    }
    static class producer extends Thread {
        public void run() {              // run method contains the thread code
            int item;
            while (true) {               // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... } // actually produce
    }
    static class consumer extends Thread {
        public void run() {              // run method contains the thread code
            int item;
            while (true) {               // consumer loop
                item = mon.remove();
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... } // actually consume
    }
}

```

```

static class our_monitor {           // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
        buffer [hi] = val;             // insert an item into the buffer
        hi = (hi + 1) % N;             // slot to place next item in
        count = count + 1;            // one more item in the buffer now
        if (count == 1) notify();     // if consumer was sleeping, wake it up
    }
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
        val = buffer [lo];             // fetch an item from the buffer
        lo = (lo + 1) % N;            // slot to fetch next item from
        count = count - 1;            // one few items in the buffer
        if (count == N - 1) notify(); // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
}

```

## 3.8. Transferencia de mensajes

- Se emplean dos primitivas:
  - Send(destino, &mensaje)
  - Receive(origen, &mensaje)
- Son llamadas al sistema, igual que monitores y semáforos.
- Pueden ser bloqueantes y no bloqueantes
- En una red se pueden perder los mensajes.
- Implementan acuses, número de mensaje, timers, etc.

- Otro problema es la autenticación
- En procesos de una sola máquina, son mensajes pequeños para el uso de registros.

En la figura siguiente:

Problema productor-consumidor con N mensajes.

```

#define N 100                                /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                               /* message buffer */

    while (TRUE) {
        item = produce_item();               /* generate something to put in buffer */
        receive(consumer, &m);              /* wait for an empty to arrive */
        build_message(&m, item);            /* construct a message to send */
        send(consumer, &m);                 /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);              /* get message containing item */
        item = extract_item(&m);            /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                /* do something with the item */
    }
}

```

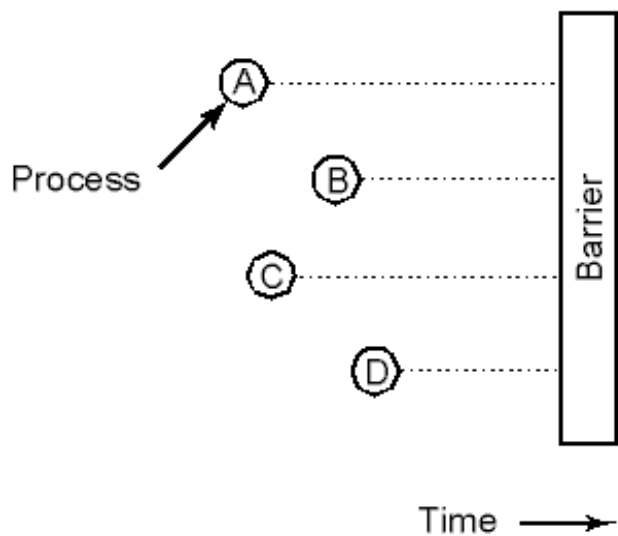
– Para la figura anterior:

- Todos los mensajes tiene el mismo tamaño
- El SO tiene un buffer para colocar los mensajes enviados y que no se han recibido.
- Intercambian el productor-consumidor (uno vacío por uno lleno)
- La cantidad de mensajes son constantes (por lo anterior)
- Emisor/receptor si uno es más rápido que el otro, con el intercambio de paquetes se regula el tráfico.
- Los obliga a operar en sincronía.
- MPI: Ejemplo de transferencia de mensajes.

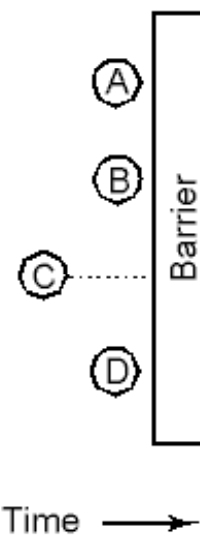


## 3.9. Barreras

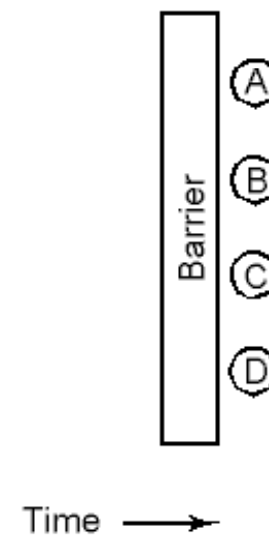
- Mecanismo de sincronización pensado para grupos de procesos.
- Cuando un proceso llega a la barrera se bloquea, hasta que todos los procesos hayan llegado a ella.
- Ver figura 2.30.
- Ejemplos: Multiplicación de matrices enormes, hamming para ráfagas, etc.



(a)



(b)



(c)

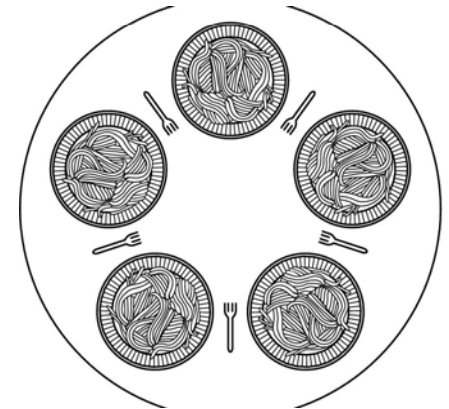
a) Procesos que se aproximan    b) Se bloquean hasta que llegue el último  
 c) Hasta que llega el último se dejan pasar a todos.

## 4. PROBLEMAS CLÁSICOS DE COMUNICACIÓN ENTRE PROCESOS

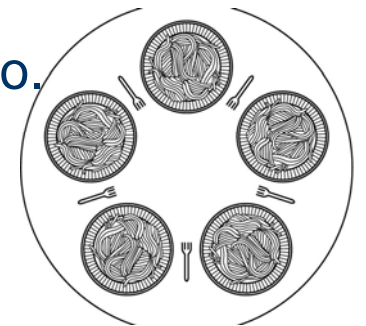
- Revisar los tres problemas:  
Ambos son problemas de sincronización  
Solucionan problemas distintos
  - a) Cena de los filósofos
  - b) Lectores – Escritores
  - c) Barbero dormilón

## 4.1. Cena de los filósofos

- Modela procesos que compiten para tener acceso exclusivo a un número limitado de recursos
- Cinco filósofos, sentados alrededor de una mesa circular.
- Cada uno tiene un plato de espagueti
- Se requieren de dos tenedores para comerlo
- Entre cada plato sólo hay un tenedor.
- Sólo comen y piensan.



- Al tener hambre:
  - Trata de tomar los tenedores (izq y der) uno a la vez en cualquier orden.
- Si logra comer, suelta los tenedores y sigue pensando.
- Primera solución:
  - Tomar tenedor izquierdo y después el derecho, o esperar a que se desocupe el derecho.
  - Podría llevar a un BI, si todos toman el izquierdo.



– Segunda solución:

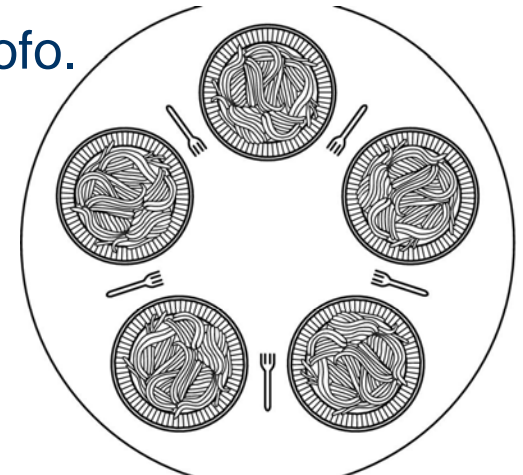
- Al tomar el tenedor izquierdo, verificar si el tenedor derecho esta libre, si no lo está debe dejar el izquierdo, esperar cierto tiempo y repetir el proceso.
- Esto podría causar inanición, podrían coincidir al volver a tomarlo.

– Tercera solución:

- Proteger con un semáforo binario.
- Antes de comenzar a tomar tenedores, el filosofo ejecuta down en mutex; al regresar los tenedores haría un up a mutex.
- Inconveniente: Sólo un filosofo podría comer.

– Cuarta solución:

- Se usa un arreglo de estados para registrar el estatus de cada filósofo en todo momento (comiendo, pensando y hambriento).
- Un filósofo comerá solo cuando ninguno de sus vecinos lo este haciendo.
- Se emplea un semáforo para cada filósofo.



```

void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                    /* enter critical region */
    state[i] = HUNGRY;                               /* record fact that philosopher i is hungry */
    test(i);                                         /* try to acquire 2 forks */
    up(&mutex);                                       /* exit critical region */
    down(&s[i]);                                     /* block if forks were not acquired */
}

void put_forks(i)                                    /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                    /* enter critical region */
    state[i] = THINKING;                            /* philosopher has finished eating */
    test(LEFT);                                     /* see if left neighbor can now eat */
    test(RIGHT);                                    /* see if right neighbor can now eat */
    up(&mutex);                                       /* exit critical region */
}

void test(i)                                         /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```



## 4.2. Lectores - Escritores

- Modela el acceso a una base de datos
- Podría haber varios lectores al mismo tiempo.
- Pero si un escritor esta en la base de datos, ningún otro proceso (escritor o lector) podrá ingresar.
- Los escritores requieren acceso exclusivo a la BD.
- Si un lector esta usando la BD y llega otro u otros lectores, estos podrían ingresar.
- Si un lector esta en la BD y llega un escritor este se suspende hasta que salga el último lector. Si llega otro lector se formaría atrás del escritor. Esto merma la concurrencia.

```

typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}

```

*/\* use your imagination \*/*  
*/\* controls access to 'rc' \*/*  
*/\* controls access to the database \*/*  
*/\* # of processes reading or wanting to \*/*

*/\* repeat forever \*/*  
*/\* get exclusive access to 'rc' \*/*  
*/\* one reader more now \*/*  
*/\* if this is the first reader ... \*/*  
*/\* release exclusive access to 'rc' \*/*  
*/\* access the data \*/*  
*/\* get exclusive access to 'rc' \*/*  
*/\* one reader fewer now \*/*  
*/\* if this is the last reader ... \*/*  
*/\* release exclusive access to 'rc' \*/*  
*/\* noncritical region \*/*

*/\* repeat forever \*/*  
*/\* noncritical region \*/*  
*/\* get exclusive access \*/*  
*/\* update the data \*/*  
*/\* release exclusive access \*/*

## 4.3. Barbero dormilón

- Modela el manejo de colas, con un mostrador de atención, y un número de clientes limitado.
- Es una barbería (peluquería)
- El establecimiento tiene un barbero, una silla para atención, y varias sillas para clientes en espera.
- Si no hay clientes presentes, el barbero se sienta y se duerme.
- Al llegar un cliente debe despertar al barbero.
- Si llegan clientes mientras atiende a alguno de ellos, estos se sentarán en una silla vacía (si las hay) o abandonarán el establecimiento.



- Utiliza tres semáforos:
  - Clientes. Clientes en espera.
  - Barberos. Número de peluqueros (0 y 1).
  - Mutex. Controla la exclusión mutua.
- Usa una variable (espera) para tener una copia de clientes (no hay manera de leer el valor actual de un semáforo).
- Una vez terminado el corte, el cliente sale, no hay ciclo como los problemas anteriores (excepto el barbero).

```

#define CHAIRS 5                                /* # chairs for waiting customers */

typedef int semaphore;                          /* use your imagination */

semaphore customers = 0;                        /* # of customers waiting for service */
semaphore barbers = 0;                         /* # of barbers waiting for customers */
semaphore mutex = 1;                           /* for mutual exclusion */
int waiting = 0;                               /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);                       /* go to sleep if # of customers is 0 */
        down(&mutex);                           /* acquire access to 'waiting' */
        waiting = waiting - 1;                  /* decrement count of waiting customers */
        up(&barbers);                            /* one barber is now ready to cut hair */
        up(&mutex);                             /* release 'waiting' */
        cut_hair();                             /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);                               /* enter critical region */
    if (waiting < CHAIRS) {                    /* if there are no free chairs, leave */
        waiting = waiting + 1;                 /* increment count of waiting customers */
        up(&customers);                         /* wake up barber if necessary */
        up(&mutex);                             /* release access to 'waiting' */
        down(&barbers);                         /* go to sleep if # of free barbers is 0 */
        get_haircut();                         /* be seated and be serviced */
    } else {
        up(&mutex);                             /* shop is full; do not wait */
    }
}

```



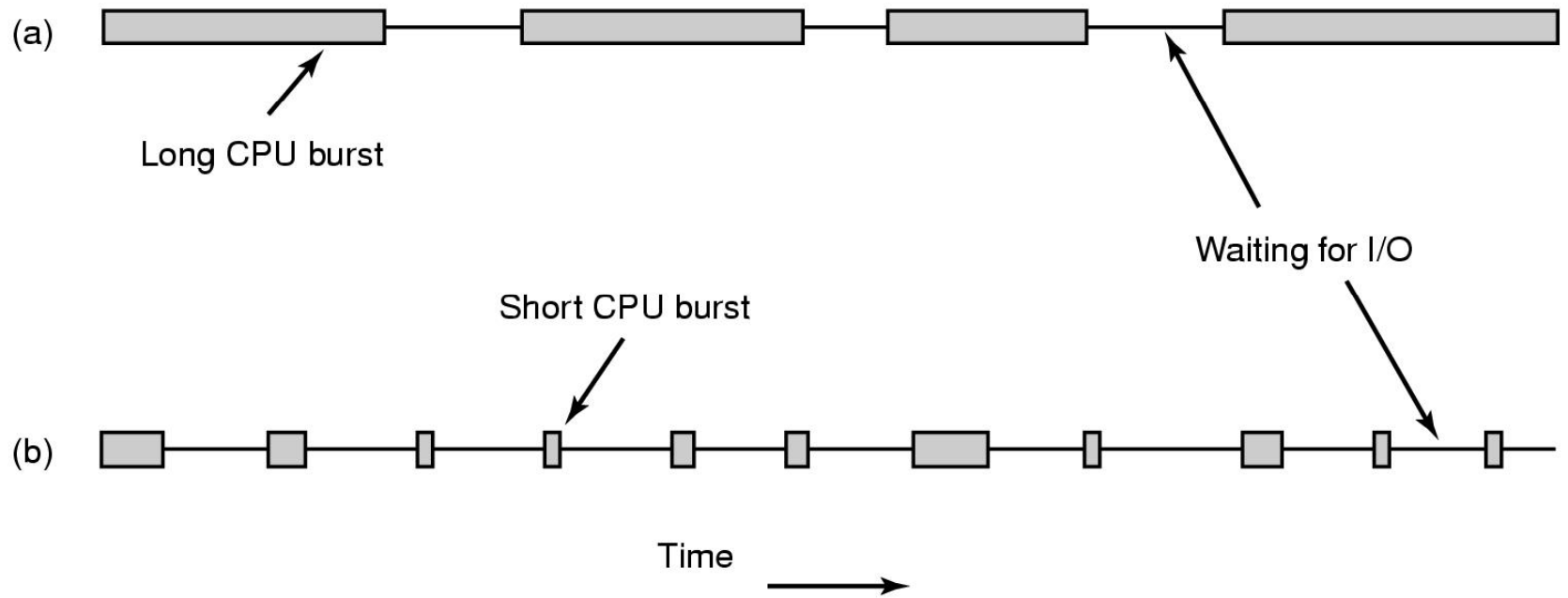
# 5. CALENDARIZACIÓN

## 5.1. Introducción a la Calendarización

- En procesamiento por lotes:
  - No hay calendarización o es muy simple.
- En sistemas de tiempo compartido:
  - Requiere algoritmos complejos
- Con las PCs:
  - Hay pocos procesos candidatos (pocas aplicaciones)
  - Las CPUs son rápidas, ya no es recurso escaso.



- Hay procesos devoradores de CPUs en PC:
  - Generación de video NTSC o PAL.
- En estaciones de trabajo y servidores:
  - Varios procesos compiten por CPU
  - Objetivos:
    - Prioridad de procesos
    - Eficientar el uso de la CPU
      - La carga/descarga de procesos es costosa.



Ráfagas de CPU alternadas con periodos de espera de E/S.  
a) Proceso dedicado a la CPU b) Proceso dedicado a E/S

- Hay procesos que se la pasan computando, mientras otros dedican más a E/S.
- ¿Cuándo calendarizar?
  1. Al crear un proceso
    - Decidir si entra el padre o el hijo
  2. Al terminar un proceso
    - Antes de agotar su tiempo. Se debe escoger otro listo o un inactivo (virtual)
  3. Al bloquearse un proceso
  4. Al recibir una interrupción
    - Debe decidir cual sacar/meter una vez terminada la interrupción

## ● Tipos de calendarización

### a) Expropiativa

- Le da un tiempo a cada proceso, si agota el tiempo y no acaba se suspende, y se toma otro listo.
- Requiere interrupciones de reloj.

### b) No expropiativa

- El proceso acapara el CPU hasta que se bloquea, acaba o lo ceda voluntariamente.
- No requiere interrupciones de reloj.

- **Categorías de algoritmos de calendarización**
  1. **Por lotes**
    - Se recomienda la calendarización no expropiativa
  2. **Interactivos**
    - Se requiere la expropiativa, para que no acaparen el CPU (ni procesos maliciosos lo hagan)
  3. **Tiempo real**
    - Casi no se requiere la expropiación, porque son procesos que actúan sobre una sola aplicación, y pueden tardar mucho algunos procesos.

- Metas de los algoritmos:

- **Todos los sistemas**

- Equidad – dar a cada proceso una porción equitativa del tiempo de CPU.
- Cumplimiento de políticas – Que se ponga en práctica la política establecida.
- Equilibrio – mantener ocupadas todas las partes del sistema, CPU y E/S.

- **Sistemas por lotes**

- Rendimiento – procesar el máximo de trabajos por hora.
- Tiempo de retorno – reducir al mínimo el lapso entre la presentación y la terminación de un trabajo.
- Utilización de CPU – mantener ocupada todo el tiempo a la CPU.

- **Sistemas interactivos**

- Tiempo de respuesta – responder rápido a las solicitudes.
- Proporcionalidad – satisfacer las expectativas de los usuarios en tiempo.

- **Sistemas en tiempo real**

- Cumplir los plazos – evitar la pérdida de datos.
- Predecibilidad – evitar la degradación de la calidad en sistemas multimedia.

## 5.2. Calendarización en sistemas por Lotes

1. FIFO
2. Trabajo más corto primero
3. Tiempo restante más corto primero
4. Calendarización de tres niveles

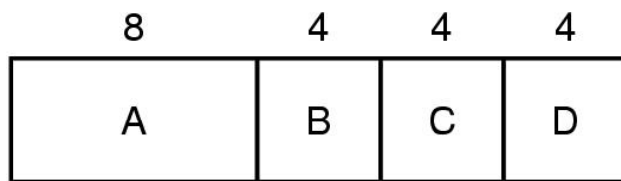
## 1. FIFO

- FIFO: Primero en llegar primero en ser atendido
- Es no expropiativo
- Hay una sola de cola de listos
- Es equitativo
- No recomendable para procesos muy heterogéneos en tiempo de CPU requerido.



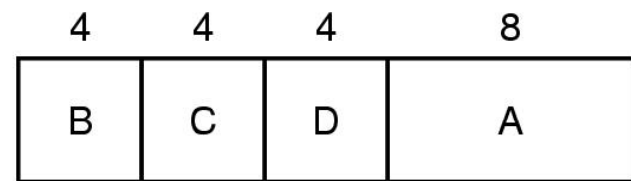
## 2. Trabajo más corto primero

- No expropiativo
- Requiere saber los tiempos de ejecución por anticipación
- Sólo funciona si están todos los trabajos disponibles.



(a)

Tiempos de retorno:  
 A=8, B=12, C=16 y D=20  
 Promedio: 14



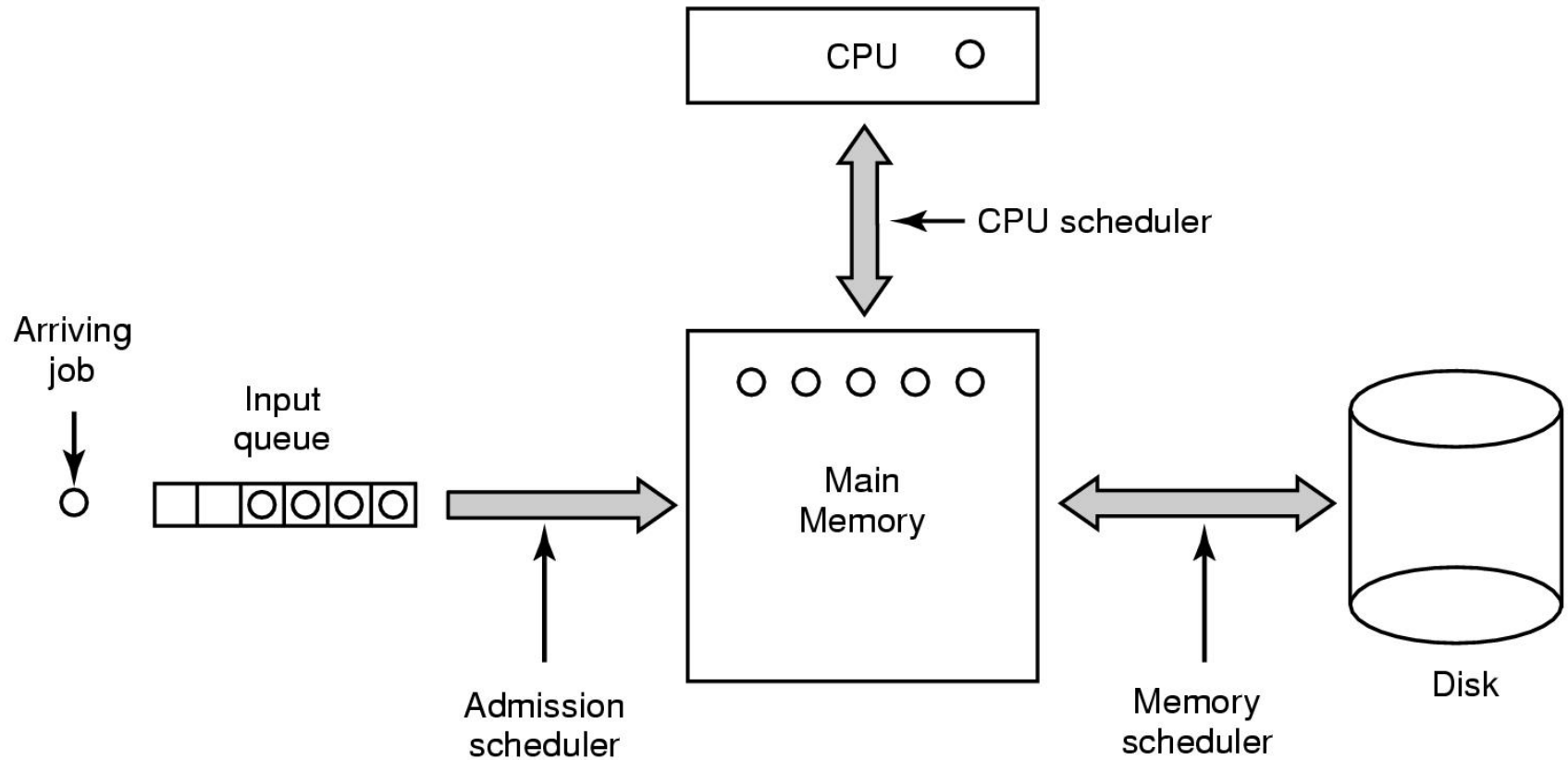
(b)

Tiempos de retorno:  
 A=4, B=8, C=12 y D=20  
 Promedio: 11

### 3. Tiempo restante más corto primero

- Requiere conocer con anticipación los tiempos de ejecución.
- Es la versión expropiativa del método anterior.
- Trabajos cortos reciben buen servicio.
- Ejemplo: Caja rápida de un banco.

## 4. Calendarización de tres niveles



### a) Calendarizador de admisión

- Decide que trabajos admitirá
  - Podría ser equilibrada de trabajos a CPU y a E/S.
  - Podría ser los trabajos más cortos primero.
  - Etc

### b) Calendarizador de memoria

- Cuando no caben los procesos en memoria principal se llevan a disco.
- Decide que procesos se quedan en memoria principal y cuales se van a disco.
- Determina el grado de multiprogramación (cuantos procesos en memoria)
- Ejemplo: Si un proceso consume el 20% de CPU se podrían tener 5.

### c) Calendarizador de CPU

- Determina cual de los procesos que están en memoria principal se ejecutará.
- Aquí se puede emplear cualquier algoritmo de calendarización.

## 5.3. Calendarización en sistemas interactivos

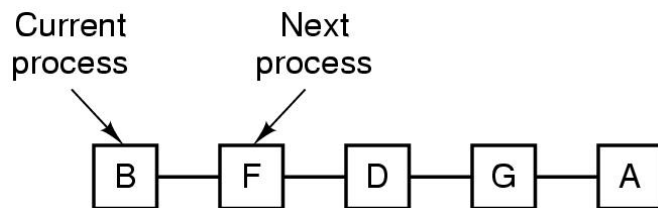
1. Calendarización por turno circular
2. Calendarización por prioridades
3. Múltiples colas
4. Proceso más corto primero
5. Calendarización garantizada
6. Calendarización por lotería
7. Calendarización por porción equitativa

## 1. Calendarización por turno circular

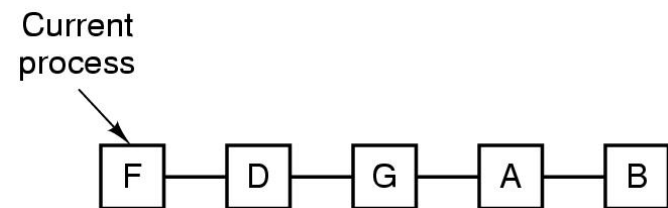
- Llamado Round-Robin
- A cada proceso se le asigna un cuanto
  - Cuanto = Intervalo de tiempo
- Es una lista simple de procesos en ejecución
- Al agotársele el cuanto pasan al final de la lista
- Aquí el detalle es el tamaño del cuanto:
  - La conmutación de paquetes requiere un gasto administrativo



- Ejemplo:
  - Cuanto de 4 ms
  - Gasto administrativo de 1 ms
  - Estaría consumiendo el 20% en gastos administrativos.
- En cuantos grandes podría haber mucha espera.
- Cuanto aceptable sería entre 20 a 50 ms.



(a)

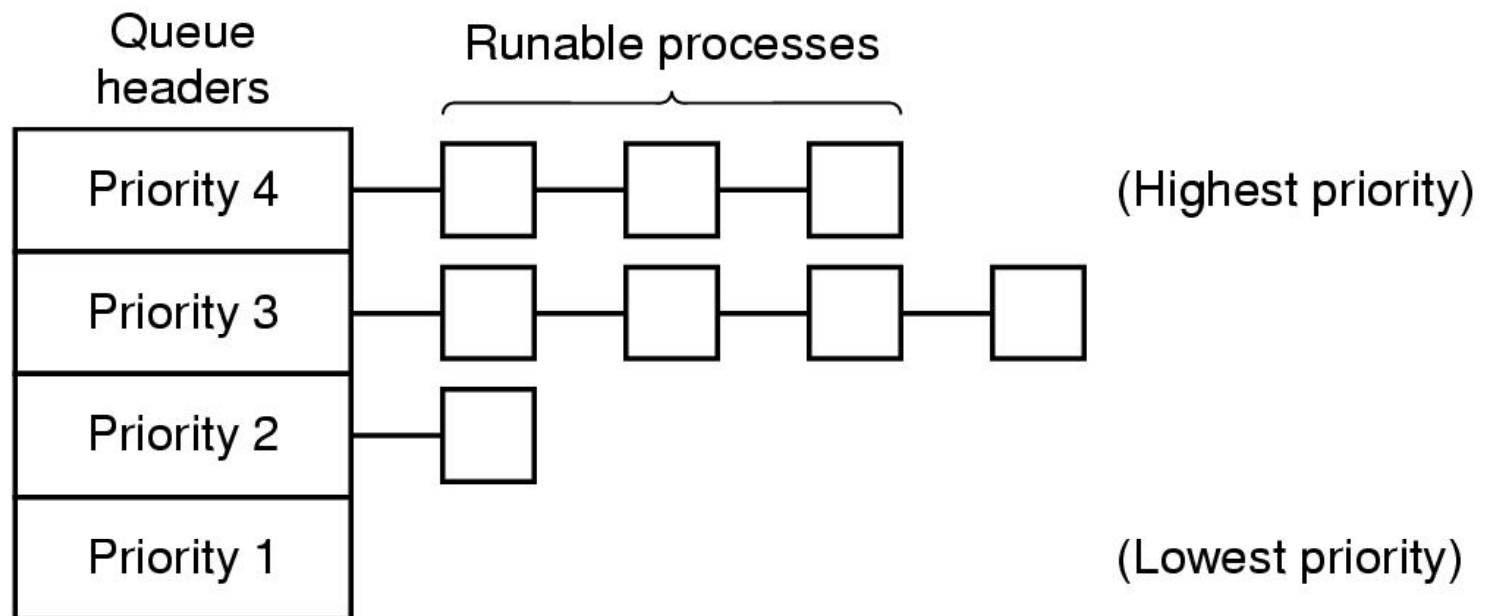


(b)

## 2. Calendarización por prioridades

- A cada proceso se le da una prioridad
- De los listos, el de mayor prioridad se ejecuta primero.
- A cada proceso se le da un cuanto máximo
- Al estarse ejecutando su prioridad debe ir decrementandose.
- Comando nice de Unix, baja la prioridad de un proceso.

- Se debe garantizar que la clase más baja no muera de inanición.



### 3. Múltiples colas

- Se establecen clases de prioridad
- Los de la clase más alta se ejecutan durante un cuanto, los de la siguiente clase en 2 cuantos, los de la siguiente clase en 4 cuantos, etc.
- Cada que un proceso se agota sus cuantos pasan a la clase inferior.
- Ejemplo:
  - Proceso que requiere un total de 100 cuantos.
  - Se ejecuta: 1, 2, 4, 8, 16, 32, 64 (37), usa 7 intercambios.
  - Al ir avanzando de clase, haría intercambios con menor frecuencia.

#### 4. Proceso más corto primero


- Es difícil saber que proceso es el más corto
- Se pueden estimar tiempos en base a comportamientos anteriores.
- Se ejecuta primero el proceso con tiempo estimado menor.
- Se emplea el envejecimiento
  - Técnica que consiste en estimar el siguiente valor de una serie, calculando la media ponderada del último valor medio y el estimado anterior
- Requiere análisis matemático.

## 5. Calendarización garantizada

- Si hay  $n$  procesos cada uno debe recibir  $1/n$  uso de CPU.
- Se requiere saber cuanto tiempo CPU ha recibido.  
 **$TD=TC/n$   $TR=Tcon/TD$** 
  - $TD$ =Tiempo que tiene derecho
  - $TC$ =Tiempo desde su creación
  - $TR$ =Tiempo recibido
  - $TCon$ =Tiempo consumido
- El algoritmo consiste en ejecutar el proceso cuyo cociente es más bajo( $TR$ ), hasta que rebase al de su competidor mas cercano.
  - $TR=0.5$  ha recibido la mitad de tiempo que le corresponde.
  - $TR=2.0$  ha recibido el doble de tiempo que le corresponde.

## 6. Calendarización por lotería

- Entrega a los procesos “billetes de lotería” para los distintos recursos del sistema.
- Al calendarizar, se escoge un billete al azar, y quien lo obtiene entra a usar el recurso.
- A los de mayor prioridad se les dan más billetes.
- Al llegar un nuevo proceso se le dan sus billetes, por lo que inmediatamente concursa.
- Se pueden pasar billetes entre procesos cooperativos.

- 
7. Calendarización por porción equitativa
    - Si hay  $n$  usuarios en sesión cada uno debe recibir  $1/n$  la capacidad del CPU.
    - Se reparte el tiempo entre los usuarios, en lugar de los procesos.



## 5.4. Calendarización en sistemas en tiempo real

- Sistemas en tiempo real
  - Deben reaccionar en cierto tiempo
    - Lectora de CDs de audio
    - Equipos en hospitales, industrias, aeropuertos, robots, etc.
- Puede haber sucesos no calendarizables
  - Por falta de tiempo no se pueden realizar
- Hay sistemas en tiempo real:
  - a) Estrictos
  - b) No estrictos  
  - a) Periódicos
  - b) No periódicos

## 5.5. Política Vs Mecanismo

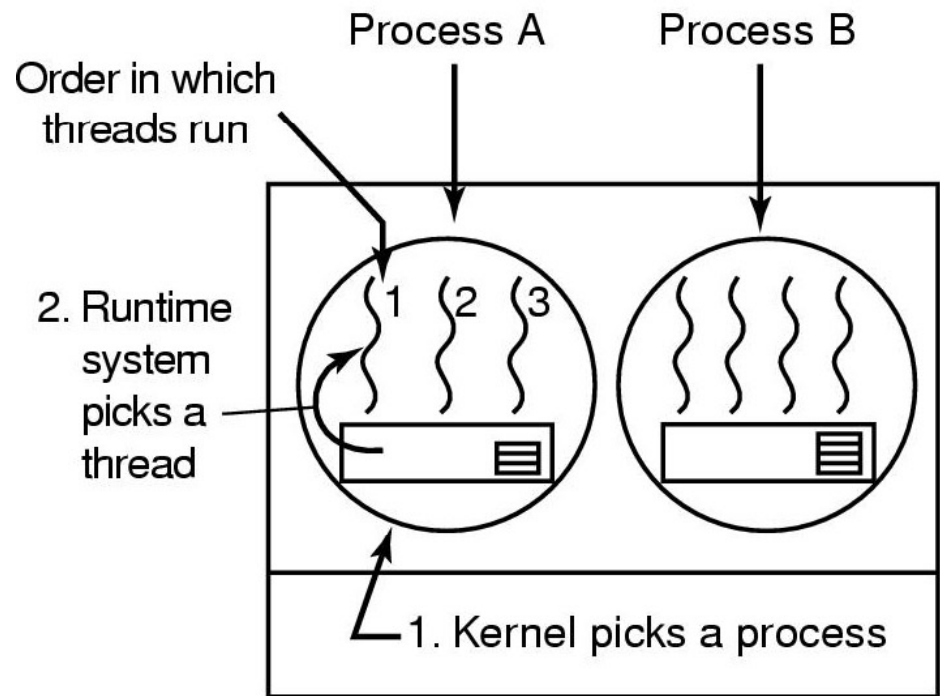
- Política
  - Usuario o proceso
- Mecanismo
  - En el kernel del SO
- Los calendarizadores deberían tomar información de los procesos para tomar la decisión óptima.

## 5.6. Calendarización de subprocesos

- Si están en modo usuario
  - El kernel no sabe de su existencia.
  - La calendarización la lleva a cabo el proceso
  - No se cuenta con un reloj calendarizador
  - Aquí podrían aplicar las políticas de los usuarios, en lugar del mecanismo del kernel.
  - Si un subproceso se bloquea, bloqueará al proceso.

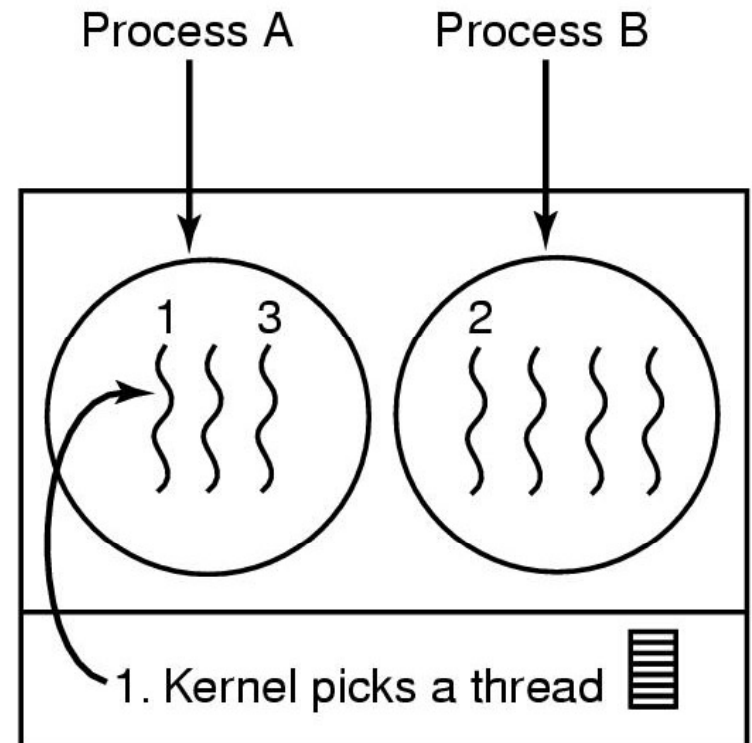
- Si están en modo Kernel
  - Se calendarizan los subprocesos sin importar a que proceso pertenece.
  - Es más costosa la carga de subprocesos en kernel.

- Calendarización de subprocesos en nivel usuario, con cuanto=50 ms, se ejecutan durante 5 ms en cada ráfaga de CPU.



Possible: A1, A2, A3, A1, A2, A3  
 Not possible: A1, B1, A2, B2, A3, B3

- Calendarización de subprocesos en modo kernel, con cuanto=50 ms, se ejecutan durante 5 ms en cada ráfaga de CPU.



Possible:      A1, A2, A3, A1, A2, A3

Also possible: A1, B1, A2, B2, A3, B3

## REFERENCIA:

- Sistemas Operativos Modernos, Segunda Edición  
TANENBAUM  
Prentice Hall