Diseño de sistemas operativos

Gestión de procesos: Una visión interna

Índice

- Introducción
- Gestión interna de eventos
- Implementación del modelo de procesos
- Operaciones sobre los procesos
- Sincronización
- Implementación de hilos

Introducción

- SO multiprogramado: servicio a programas y gestión hardware
  - Ejecución concurrente de múltiples actividades independientes
- Ejecución de un programa se intercala con:
  - Rutinas de interrupción
  - Llamadas al sistema
  - Excepciones
  - Ejecución de otros programas (multiprogramación)
- Pero procesador sólo ejecuta una actividad
  - SO reparte recursos hardware entre actividades concurrentes
- Concepto de proceso:
  - Abstracción de un "procesador virtual"
    - SO hace creer a programa que tiene una máquina dedicada

Introducción

- Programa *versus* Proceso
  - Programa (pasivo) != Proceso (activo)
  - Múltiples procesos ejecutando el mismo programa (p.e. *shell*)
  - En UNIX se aprecia claramente la diferencia:
    - FORK: Nuevo Proceso - Mismo Programa
    - EXEC: Mismo Proceso - Nuevo Programa
    - Un proceso puede ejecutar varios programas durante su vida
  - En otros sistemas proceso asociado a programa “para toda la vida”
    - Windows: *CreateProcess*: Nuevo Proceso - Nuevo Programa
- Objetivo del tema:
  - Estudiar cómo SO implementa el modelo de procesos a partir de la gestión de los **eventos** internos del procesador (interrupciones, excepciones y llamadas al sistema)

Índice

- Introducción
- Gestión interna de eventos
  - Gestión hardware de eventos
  - Gestión de eventos por el sistema operativo
- Implementación del modelo de procesos
- Operaciones sobre los procesos
- Sincronización
- Implementación de hilos

Modos de ejecución del procesador

- UCP proporciona modos de ejecución con diferentes privilegios
- SO requiere al menos 2:
  - Uno con privilegio total y otro con privilegio mínimo
  - Incluso aunque haya más (Pentium tiene 4) sólo usa 2
    - Moraleja: SO se conforma con mínimos (más transportable)
- Modo usuario (no privilegiado):
  - Acceso restringido a instrucciones, registros, E/S
  - Memoria: Sólo accesible direcciones lógicas de usuario
- Modo sistema (núcleo o privilegiado):
  - Privilegio total
- Procesador se inicia en modo sistema
  - Con interrupciones inhibidas y hardware de memoria desactivado
  - Va transitando entre ambos modos

Cambio de modo de ejecución

- UCP usa dos pilas: pila de usuario y de sistema
- Evento causa que procesador pase a modo sistema:
  - HW salva info. en pila de sistema
    - Típicamente PC y R.estado; SW (SO) salvará el resto
  - Pone UCP en modo sistema
  - Salta a rutina de tratamiento almacenada en vector de interrupción
  - Puede haber anidamiento en tratamiento de eventos
- Fin de rutina (RETI): procesador retorna a modo previo:
  - HW restaura info. salvada en pila recuperando modo previo
  - Si no anidamiento, retorna a modo usuario
- Algunos procesadores dos pilas de sistema:
  - Pila de interrupción: para interrupciones
  - Pila de sistema: para llamadas al sistema y excepciones

Interrupciones

- Notifican circunstancia que se debe atender
- Carácter asíncrono
- Generadas por:
  - Dispositivos de E/S
  - Condiciones críticas en el sistema (p. ej. fallo en el hardware)
- Múltiples líneas de petición de interrupción
  - Conectadas directamente a UCP o a través de controlador
  - Cada línea se identifica por vector (configurable)
  - Puede haber múltiples dispositivos en cada línea

Esquemas de gestión de interrupciones

- Esquema sin niveles de interrupción
  - Las interrupciones pueden estar permitidas o inhibidas
  - Inicialmente, inhibidas
  - Existen instrucciones para cambiar el estado de las interrupciones
  - Cuando se acepta interrupción se inhiben
  - Posibilidad de enmascarar una determinada línea
- Esquema con niveles de interrupción
  - Cada línea tiene una prioridad
  - En cada momento UCP ejecuta en un nivel (Nivel UCP)
    - Sólo admite int. de nivel > Nivel UCP
  - Cuando acepta int. de nivel N, automáticamente Nivel UCP=N
  - Al terminar la rutina de interrupción retorna al nivel previo
  - Inicialmente, UCP en nivel máximo
  - Existen instrucciones para cambiar nivel de UCP

Interrupciones en un multiprocesador

- Modalidad de distribución de interrupciones configurable
  - Fija; *Broadcast*; *multicast*;
  - Turno rotatorio; por prioridad; a UCP más reciente; ...
- Inicialmente, sólo procesador maestro arrancado
  - Distribución de interrupciones fija: al maestro
- Interrupción entre procesadores (IPI)
  - Usada para planificación, gestión de memoria, ...
- Recuerde: inhibir interrupciones sólo afecta a UCP local

Excepciones

- Situaciones de carácter excepcional al ejecutar una instrucción
- Carácter síncrono
- Normalmente, errores de programa:
  - dividir por cero, instrucción privilegiada, ...
- Pero no siempre:
  - Fallos de página; excepciones de depuración
- Modo previo: usuario o sistema
- No cambia el estado de las interrupciones
- Vector predefinido
  - Intel vector 0: excepción de división por cero

Llamadas al sistema

- Instrucción no privilegiada causa llamada
  – En Pentium INT o SYSENTER/SYSEXIT (menos sobrecarga)
- Carácter síncrono
- Modo previo: Usuario
- No cambia el estado de las interrupciones
- Puede usar un vector (fijo o un operando de la instrucción)
  – En Linux/Pentium: INT 0x80 (vector 0x80)
  – En Windows/Pentium: INT 0x2e (vector 0x2e)
- O no usarlo:
  – SYSENTER usa un registro privilegiado

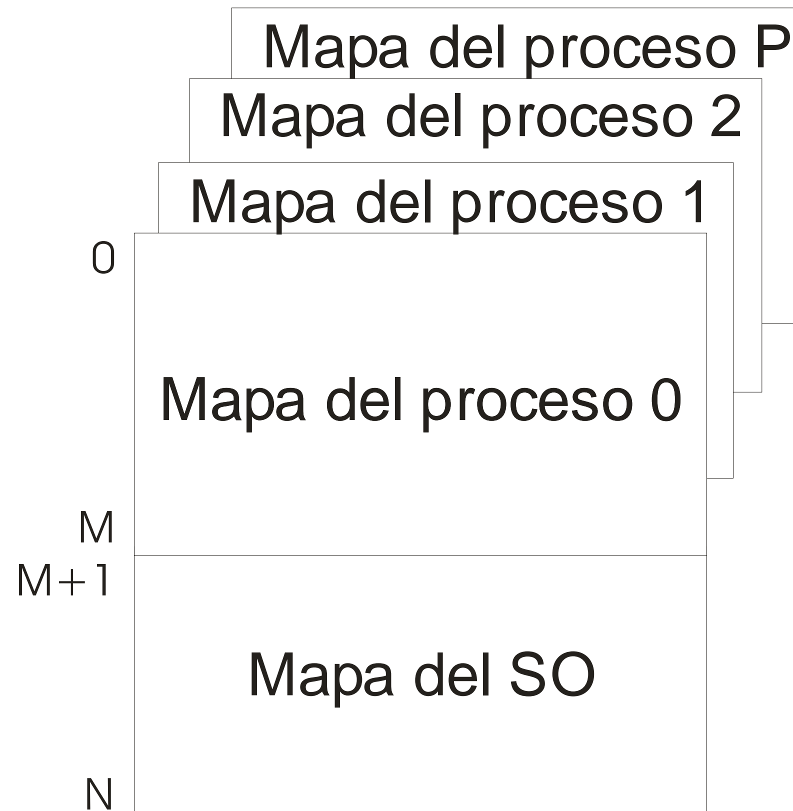
Programa dirigido por eventos

```
void tratar_boton_OK(){}
void tratar_boton_Cancel(){}
int main(){
    inicia_estructuras_de_datos();
    establecer_manejador(boton_Cancel, tratar_boton_Cancel);
    establecer_manejador(boton_OK, tratar_boton_OK);
    ........
    presentar_ventana_inicial();
    pausa();        // se queda indefinidamente bloqueado a la
                    // espera de eventos
}
```

Gestión de eventos por el SO

- SO dirigido por eventos:
  - Tabla de vectores de interrupción: direcciones de rutinas del SO
- Fase inicial del SO (modo sistema e interrupciones inhibidas)
  - Inicia HW, tabla de vectores y sus estructuras de datos
    - En MP: configura distribución de int. y arranca otras UCP
  - Crea proc. inicial tal que se ejecute en m. usuario e int. habilitadas
  - SO cede control al proceso inicial
- UCP sólo vuelve a modo sistema por evento:
  - Tabla de vectores asegura que ejecute SO

Modelo de memoria del proceso



Esquemas de gestión de eventos

- Primera aproximación a modo de operación del SO
  - Programa en ejecución (m. usuario); se produce un evento
  - SO lo trata usando pila de sistema (puede haber eventos anidados)
  - Fin de tratamiento: retorna a modo usuario y continúa proceso
    - Puede ser el que fue interrumpido u otro
- No hay cambio de proceso en mitad del tratamiento de evento

- Las cosas no son tan fáciles. Veamos un ejemplo.

Ejemplo: solución errónea

```
char *dir_buf; // guarda la dirección donde se copiará el dato leído
int lectura(char *dir) {
    dir_buf = dir;
    out(R_CONTROL, LECTURA); // programa el dispositivo
    Retorna cediendo el control a otro proceso;
}
void interrupcion() {
    *(dir_buf) = in(R_DATOS); // lee el carácter y lo copia
    Marca que el proceso lector ya puede ejecutar;
    Si dicho proceso es más prioritario, retorna al mismo;
    En caso contrario retorna al proceso interrumpido;
}
```

Ejemplo: traza errónea

- P1 solicita leer: *dir=100*

Planteamiento de una solución

- El propio proceso lector debe realizar la copia
- Una vez terminada la interrupción debe proseguir la llamada
- Si la llamada debe esperar, se salva estado en ese punto
  - Información de parámetros y variables en la pila de sistema
- Cuando se reanude, continuará justo en ese punto
- Llamada tiene fases:
  - 2 en el ejemplo, pero puede tener un nº considerable (*open*)
- Hay cambio de proceso en mitad del tratamiento de evento
  - Sólo si llamada o excepción; nunca con interrupción
  - Debe haber una pila de sistema por proceso
- Múltiples tratamientos pendientes de completar en cada momento
  - **Modelo de procesos**: SO es pasivo, procesos entidades activas

Ejemplo: solución válida

```
char buf; // guarda el dato leído
int lectura(char *dir) {
    out(R_CONTROL, LECTURA); // programa el dispositivo
    Guarda estado en ese punto y cede control a otro proceso;
    *dir = buf; // continuará ejecutando esta sentencia
}
void interrupcion() {
    buf = in(R_DATOS); // lee el carácter y lo copia
    Marca que el proceso lector ya puede ejecutar;
    Si dicho proceso es más prioritario, cede control al mismo;
    En caso contrario retorna al proceso interrumpido;
}
```

Ejemplo: traza válida

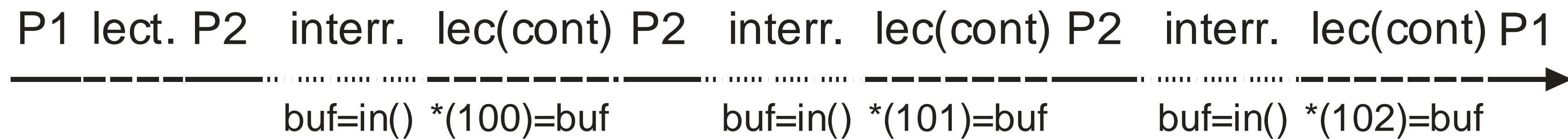
- P1 solicita leer: *dir=100*

Ejemplo: múltiples fases

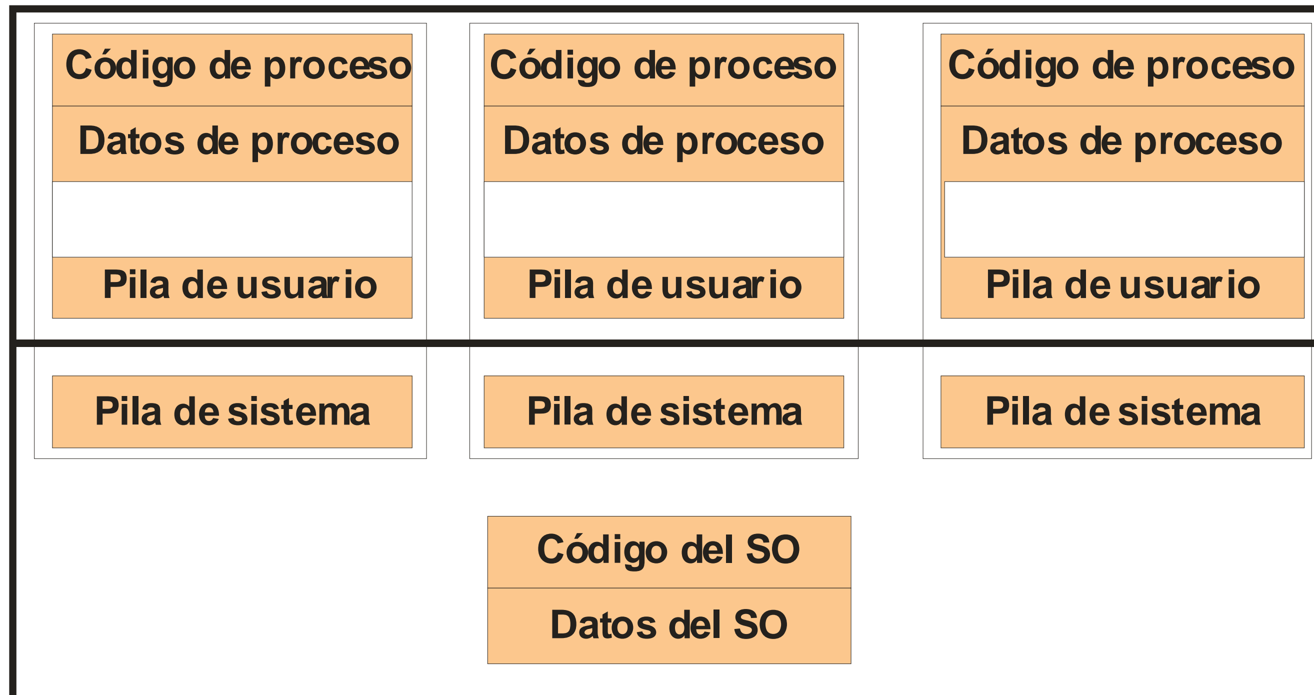
```
char buf; // guarda el dato leído
int lectura(char *dir, int tam) {
    while (tam--) lee_caracter(dir++); }
void interrupcion() {
    buf = in(R_DATOS); // lee el carácter y lo copia
    Marca que el proceso lector ya puede ejecutar;
    Si dicho proceso es más prioritario, cede control al mismo;
    En caso contrario retorna al proceso interrumpido; }
int lee_caracter(char *dir) {
    out(R_CONTROL, LECTURA); // programa el dispositivo
    Guarda estado en ese punto y cede control a otro proceso;
    *(dir) = buf; // continuará ejecutando esta sentencia }
```

Ejemplo: traza múltiples fases

- P1 solicita leer: *dir=100; tam=3*



Modelo de procesos



1 pila de sistema/proceso

El modelo de interrupciones

- Inconveniente de modelo de procesos: 1 pila de sistema/proceso
  - Problemas de *escalabilidad*
- Esquema alternativo: modelo de interrupciones
  - No hay cambio de proceso en mitad del tratamiento de evento
  - Basta con una pila de sistema/proceso (realmente 1/procesador)
  - Llamada que no completa trabajo, declara **continuación** y termina
  - Necesidad de almacenar datos para la función de continuación
  - Varias fases: continuación especifica otra continuación
  - Modo de programación poco natural y propenso a errores
  - Más propicio en SO con arquitectura de micronúcleo (QNX)
    - Aunque L4 usa modelo de procesos
    - Mach permite usar ambos modelos
- Resto de exposición basada en modelo de procesos

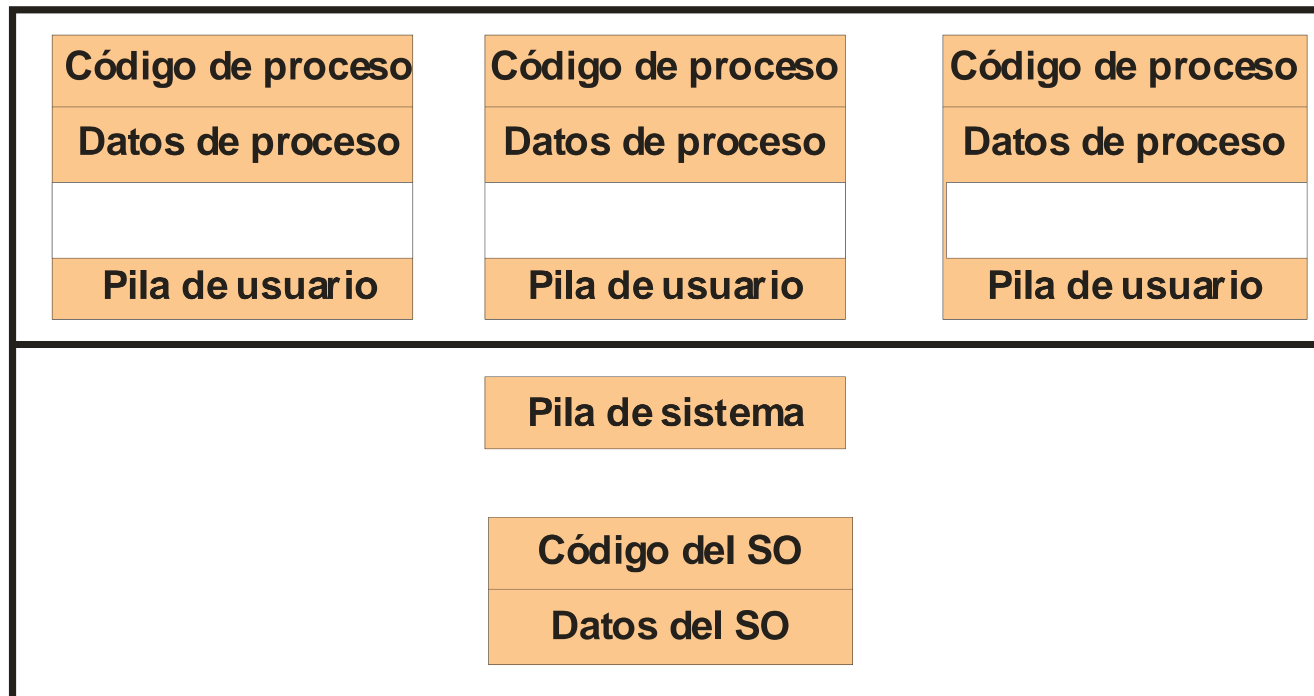
Ejemplo: modelo de interrupciones

```
char buf; // guarda el dato leído
int lectura(char *dir, int tam) {
    out(R_CONTROL, LECTURA); // programa el dispositivo
    Establece “cont_lectura” como continuación; “dir” y “tam” → estructura de datos;
    Retorna cediendo el control a otro proceso; }
int cont_lectura() {
    char *dir_aux; int tam_aux; ← valores recuperados desde estructura de datos;
    *(dir_aux++) = buf;
    if (--tam_aux>0) {
        out(R_CONTROL, LECTURA); // programa el dispositivo
        Establece “cont_lectura” como continuación; “dir_aux” y “tam_aux” → e. datos;
        Retorna cediendo el control a otro proceso; }
    else    Retorna al proceso que hizo la llamada al sistema; }
void interrupcion() {
    buf = in(R_DATOS); // lee el carácter y lo copia
    Marca que proceso lector puede ejecutar y si más prioritario, retorna a su continuación;
    En caso contrario retorna al proceso interrumpido; }
```

Ejemplo: traza modelo interrupciones

- P1 solicita leer: *dir=100; tam=1*

Modelo de interrupciones

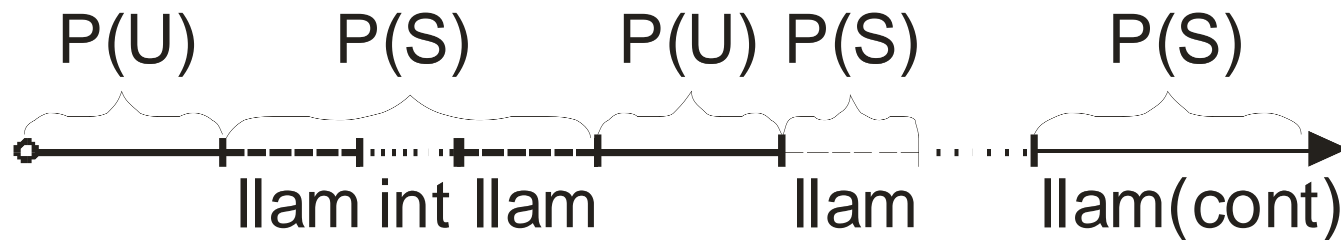


1 pila de sistema/procesador

Modos de ejecución de un proceso

- Tratamiento de un evento causa cambio de modo pero:
  - NO cambio de contexto: se realiza en contexto del proceso activo
    - Mapa de mem. activo corresponde con proceso en ejecución
    - Incluso aunque no tenga nada que ver con el evento
- SO se puede considerar entidad pasiva (como una biblioteca)
  - SO no es un proceso: crea la abstracción de proceso
  - Proceso única entidad activa
  - Excepto en inicio, siempre proceso ejecutando (aunque sea el nulo)
- Modelo: código del SO lo ejecutan los procesos
- Proceso con dos modos de ejecución: usuario y sistema
  - Modo usuario: Ejecutando el programa correspondiente
  - Modo sistema: Ejecutando el SO (rutina tratamiento de evento)

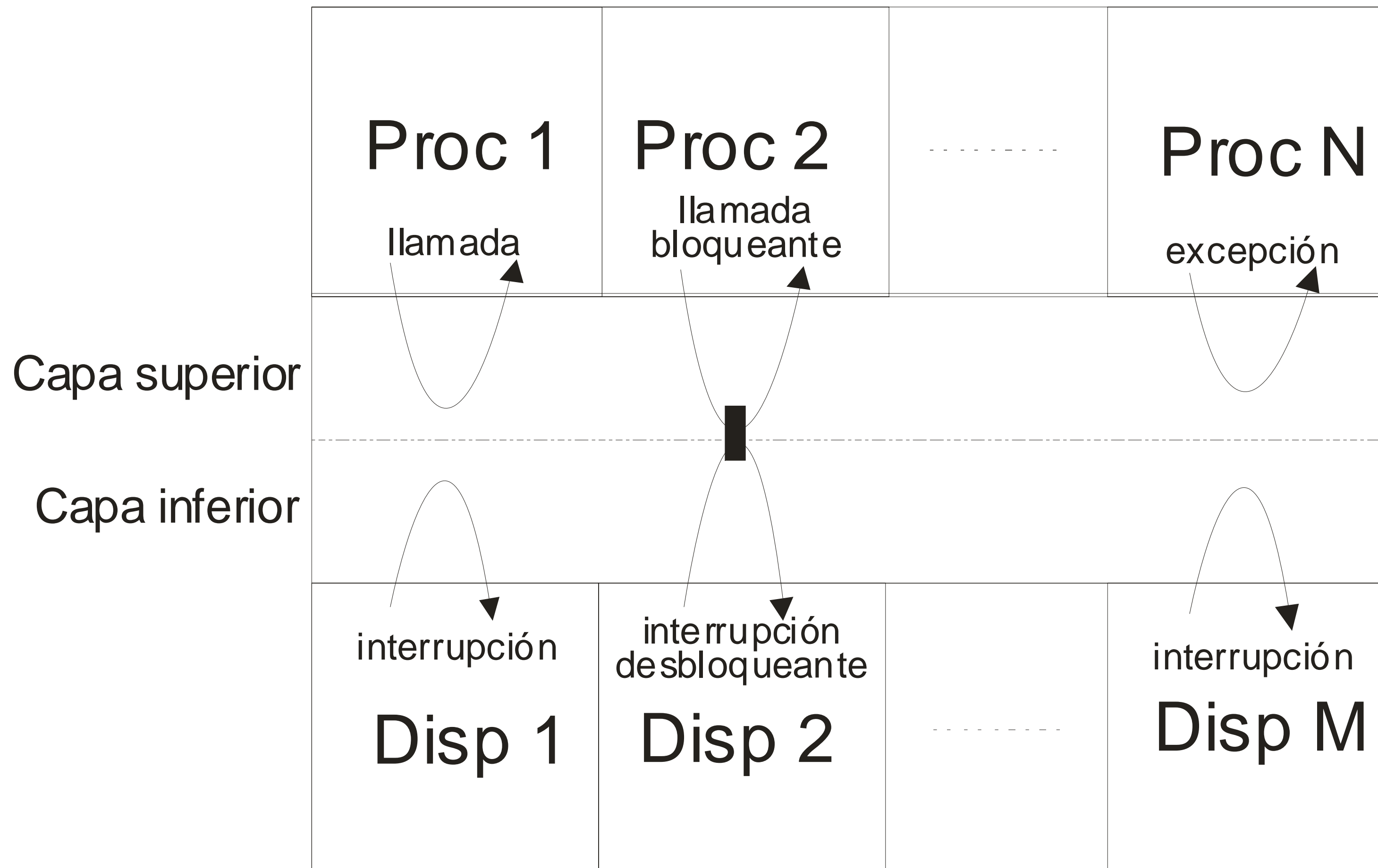
Traza de proceso con cambios de modo



Tratamiento de eventos síncronos/asíncr.

- Eventos síncronos (llamada al sistema o excepción)
  - Ejecución en el contexto del proceso "solicitante"
  - Se puede acceder a mapa de memoria de proceso actual
  - Se puede realizar una operación de bloqueo
- Actividades asíncronas (interrupción)
  - Ejecución en contexto de proceso no relacionado con la interrup.
  - No se puede acceder a mapa de memoria de proceso actual
  - No se puede realizar una operación de bloqueo
- SO puede considerarse dividido en 2 partes:
  - Capa superior que trata eventos síncronos
    - Más "pegada" a los procesos
  - Capa inferior que trata eventos asíncronos
    - Más "pegada" a los dispositivos

Organización del SO



Ejercicio

- Buscar errores:
  - Versión 1: a vs. b
  - Versión 2: a vs. b
- ¿Diferencia entre versión 1 y versión 2?

Versión 1a

```
tipo_lista_proc espera;
tipo_lista_proc list_opera;
dato *destino;
LeerX (dato *dir_usu) {
    Si (list_opera!= NULL)
        Bloquear(espera);
    Programar_disp_X();
    destino=dir_usu;
    Bloquear(list_opera);
    Si (espera != NULL)
        Desbloquear(espera);
}
```

```
InterrupciónX {
    *destino=reg_datos;
    Desbloquear(list_opera);
}
```

Versión 1b

```
tipo_lista_proc espera;
tipo_lista_proc list_opera;
dato buf_int;
LeerX (dato *dir_usu) {
    Si (list_opera!= NULL)
        Bloquear(espera);
    Programar_disp_X();
    Bloquear(list_opera);
    *dir_usu=buf_int;
    Si (espera != NULL)
        Desbloquear(espera);
}
```

```
InterrupciónX {
    buf_int=reg_datos;
    Desbloquear(list_opera);
}
```

Versión 2a

```
tipo_lista_proc lista;
tipo_cola_peticiones cola;
LeerX (void *dir_usu) {
    dato *p_buf_int;
    p_buf_int=ResMem();
    InserPet(cola,p_buf_int);
    Si (lista != NULL)
        Bloquear(lista);
    Sino {
        Programar_disp_X();
        Bloquear(lista);
    }
    *dir_usu=*p_buf_int;
    LibMem(p_buf_int);
}
```

```
InterrupciónX {
    dato *p_buf_int;
    p_buf_int=ExtrPet(cola);
    *p_buf_int=reg_datos;
    Desbloquear(lista);
    Si (lista != NULL)
        Programar_disp_X();
}
```

Versión 2b

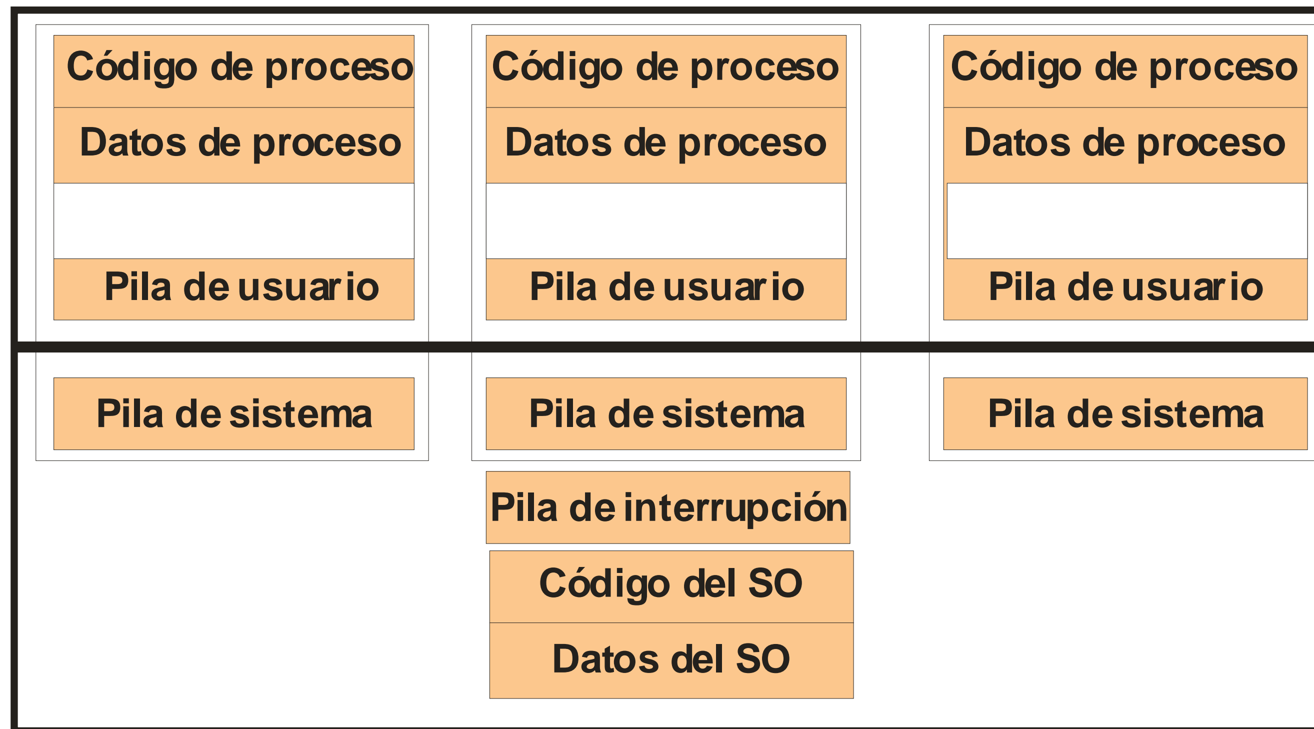
```
tipo_lista_proc lista;
tipo_cola_peticiones cola;
LeerX (void *dir_usu) {
    dato *p_buf_int;
    p_buf_int=ResMem();
    InserPet(cola,p_buf_int);
    Si (lista != NULL)
        Bloquear(lista);
    Sino {
        Programar_disp_X();
        Bloquear(lista);
    }
    *dir_usu=*p_buf_int;
    LibMem(p_buf_int);
}
```

```
InterrupciónX {
    dato *p_buf_int;
    p_buf_int=ExtrPet(cola);
    *p_buf_int=reg_datos;
    Desbloquear(lista);
    Si (lista != NULL) {
        Programar_disp_X();
        Bloquear(lista);
    }
}
```

Gestión de pilas del sistema

- Si procesador gestiona sólo una pila de sistema:
  - 1 pila del sistema/proceso
  - Reservar tamaño para máximo anidamiento de eventos
    - Llamada – excepción – todas las interrupciones
- Si procesador gestiona pila de sistema y de interrupción:
  - Pila de sistema (eventos síncronos): máximo anidamiento
    - Llamada – excepción
  - Pila de interrupción (eventos asíncronos): máximo anidamiento
    - Todas las interrupciones
  - Tratamiento de e. síncrono con varias fases, pero no e. asíncrono
    - 1 pila de sistema/proceso y 1 sólo 1 de interrupciones (1/UCP)
- Uso de pila de sistema y de interrupción también por software

Modelo de procesos con 2 pilas de sistema



1 pila de sistema/proceso | 1 pila de interrupción/procesador

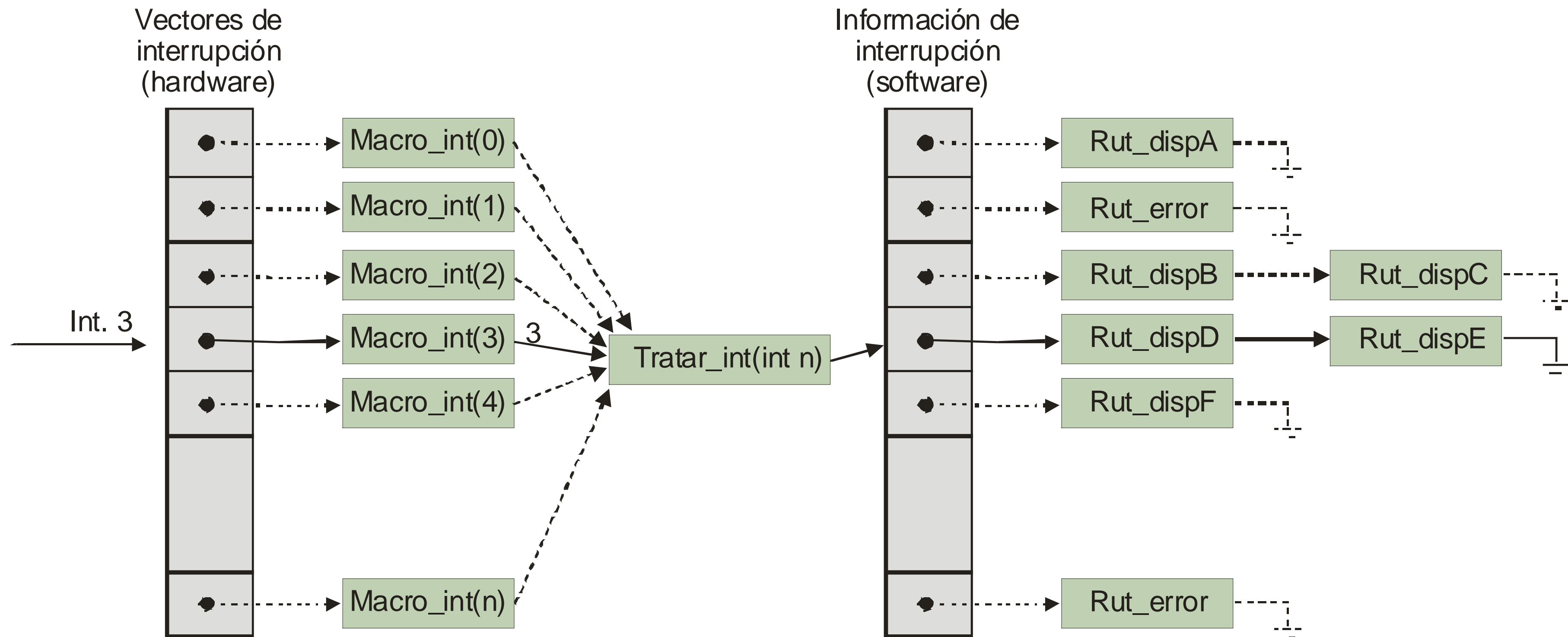
Tratamiento de interrupciones

- Modelo de gestión de SO independiente del procesador
  - Windows usa esquema con prioridades aunque Intel no lo tenga
  - Linux usa esquema sin prioridades aunque SPARC lo tenga
- Tipos de operaciones asociadas a una interrupción
  - Críticas: ejecución inmediata con interrupciones inhibidas
  - Urgentes: resultado erróneo si llega interrupción del mismo tipo
  - No urgentes: ejecutan con todas las interrupciones permitidas
- Implementación en un procesador con esquema sin prioridades:
  - Críticas: al principio de la rutina con interrupciones inhibidas
  - Urgentes: dentro de la rutina de interrupción con int. habilitadas
    - Pero las del mismo tipo enmascaradas
  - No urgentes: ejecutadas fuera del ámbito de rutina de interrupción
    - Ejecución diferida mediante **interrupción software**

Tratamiento de interrupciones

- Esquema de tratamiento debe:
  - Minimizar ensamblador
  - Factorizar código
  - Permitir múltiples rutinas de interrupción por cada línea
- Vector contiene dirección de macro en ensamblador:
  - Salva registros en pila de sistema
  - Invoca función genérica pasándole identificador de interrupción
  - Restaura registros de la pila del sistema del proceso y RETI
- Función genérica (común a todas las interrupciones)
  - Invoca función (o funciones) específica del manejador
- Función específica incluida forma parte de manejador
- Se debe minimizar tiempo con interrupciones inhibidas
  - Reduce **latencia** en el tratamiento de la interrupción
    - Desde que se activa hasta que comienza a ejecutar rutina

Esquema de gestión de interrupciones



Tratamiento de excepciones

- Rutina en ensamblador salva regs. e invoca rutina de alto nivel
- Si error, tratamiento depende de nivel previo de procesador
- Si era sistema: Error en código de SO (pánico)
  - Se muestra mensaje e info. de depuración
  - Se aborta proceso actual o se detiene SO
- Si era usuario: Error en programa de usuario
  - Si está siendo depurado, se notifica a depurador
  - Si el programa establece manejador, ejecutarlo
  - En caso contrario, se aborta el proceso
  - En UNIX manda señal a proceso
  - En Windows tratamiento estructurado de excepciones (como Java)
- No siempre error: Puede tratarse de un fallo de página
  - Tanto en usuario como en sistema (pero no desde interrupción)

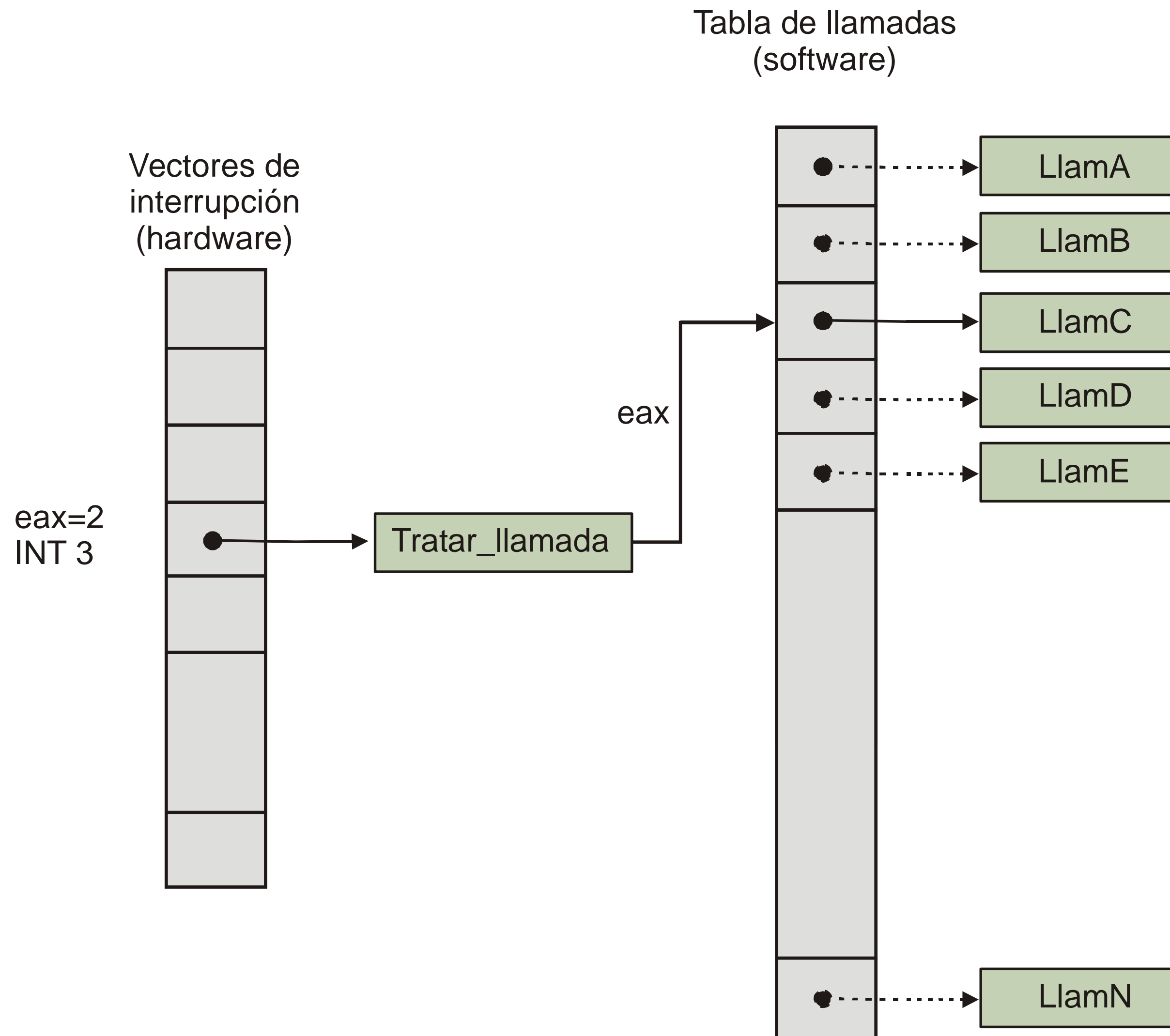
Tratamiento de llamadas

- 1 vector de int. para todas las llamadas (o ninguno SYSENTER)
- Cada llamada tiene asignado un número
- SO gestiona tabla de llamadas:
  - Vector con direcciones de rutinas que realizan cada servicio
- SO recibe nº de llamada (normalmente en un registro):
  - SO indexa con él en el vector
- Parámetros de la llamada en pila de usuario o en registros
  - Linux usa registros (como mucho 5)
  - Windows usa pila de usuario
    - Complicación: página de pila de usuario no residente
- Resultado de la llamada se suele devolver en un registro

Esquema de tratamiento de llamadas

- Estructura de la rutina de tratamiento:
  - Salva registros en pila de sistema
    - Si parámetros en registros, quedan en pila de sistema
  - Si parámetros pasados en la pila de usuario
    - Copiarlos a pila de sistema para simplificar
  - Obtener número de llamada (normalmente, en un registro).
  - Invocar función indexando en tabla de llamadas
    - la función obtiene parámetros de la llamada en pila de sistema
  - Devuelve resultado (normalmente, en un registro)
  - Restaurar registros de pila del sistema del proceso
  - RETI

Esquema de tratamiento de llamadas



Validación de parámetros

- Hay que validar parámetros de tipo puntero
- Solución pesimista (Windows):
  - Se comprueba que valores de punteros son válidos antes de usarlos
- Solución optimista (Linux):
  - Se usa directamente, si inválido → excepción en modo sistema
    - Mínima comprobación antes de uso: que no es dirección del SO
  - Ante excepción de memoria en modo sistema...
    - ¿cómo se determina que no es un error en el código del SO?
  - Linux: tabla con dir. de instrucciones de SO que acceden a paráms.
    - Instrucción que causa excepción ⊂ tabla, llamada retorna error
- Optimista más eficiente pero más complejo
  - El error se puede producir en medio de una llamada

Invocación de llamadas al sistema

- Interfaz a los programas: rutina de envoltura de biblioteca
  - *read(f, buf, tam)*
- Programa se enlaza con biblioteca de llamadas
- Rutina de llamada en ensamblador (parám. en registros)

```
close(int desc){
  MOVE R0, #NUM_CLOSE
  MOVE R1, desc
  INT 0x80
  Valor devuelto está en R0
  RET
}
```

- En UNIX: si valor devuelto <0
  - Función retorna -1 y errno = |valor devuelto|

Necesidad de diferir operaciones

- Escenario frecuente en código del SO:
  - Se detecta que hay realizar una operación pero...
  - Se debe diferir su ejecución hasta momento/contexto oportuno
- Ejemplo: interrupción de teclado
  - Rutina interrupción realiza operaciones urgentes
  - No urgentes ejecutan en contexto con interrupciones habilitadas
  - Pero cuando haya terminado posible anidamiento de interrupciones
- Ejemplo: interrupción de disco desbloquea proceso más prioritario
  - Cambio de proceso sólo al final del posible anidamiento de eventos
- Todas las soluciones similares:
  - Se anota en alguna variable del SO que hay cierto trabajo pendiente
  - Cuando contexto oportuno, consulta variable y si pendiente se hace
- Para evitar redundancia, ¿por qué no dar soporte a este requisito?
  - Interrupciones software

Interrupciones software

- Activación de una interrupción de baja prioridad por software
  – Interrupción de baja prioridad pero activada por software
  – Similar a llamada al sistema pero previamente en modo sistema
  – Consecuencia: carácter asíncrono en vez de síncrono
- Utilidad: diferir ejecución de operación hasta momento oportuno
  – Cuando haya terminado el posible anidamiento de eventos
- En ejemplos: en rutina de int. teclado o disco se activa int. SW
  – En rutina int. SW ops. no urgentes o cambio de proceso, respectiva.
- No imprescindible pero facilita tratamiento general
- Implementada por SO (algunos procesadores dan cierto soporte)
- Interrupciones software de sistema versus de proceso

Interrupciones SW de sistema y de proceso

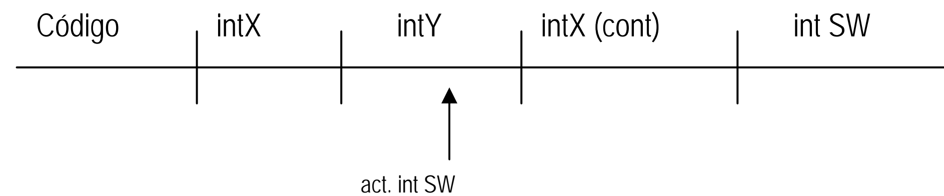
- Interrupciones SW de sistema
  - Ámbito global: no vinculadas a ningún proceso
    - Su tratamiento se difiere hasta contexto oportuno
  - Ejemplo interrupción de teclado
    - Rutina interrupción de teclado realiza operaciones urgentes
    - Y activa interrupción software de sistema
    - Tratamiento de interrupción SW de sistema → ops. no urgentes
- Interrupciones SW de proceso
  - Dirigida a un proceso *P*
    - Puede ejecutar en UCP ≠ donde se activa int. SW (o no ejecutar)
    - Su tratamiento se difiere hasta contexto oportuno y *P* ejecute
  - Ejemplo: interrupción de disco desbloquea proceso
    - Si más prioritario que algún proceso en ejecución
    - Enviar interrupción SW de proceso a proceso menos prioridad
    - Tratamiento de interrupción SW de proceso → cambio de proceso

Interrupciones software de sistema

- Ejecución diferida de operaciones no urgentes de int. dispositivos
- Implementación: variable guarda si int. SW pendiente
  - Se consulta variable cuando contexto oportuno
- Número de int. SW de sistema disponibles:
  - Sólo 1 (DPC de Windows)
    - Lista de tareas pendientes asociadas a int. SW de sistema
  - Múltiples (1 por dispositivo) con distintas prioridades
    - Implementación: 1 bit por cada int. SW de sistema
    - *Softirqs* de Linux (10): temporizador, reloj, etc.
    - Aunque 2 *softirqs* destinadas a encolar tareas pendientes (*tasklets*)
- Características de int. SW de sistema:
  - No acceso a mapa de proceso
  - No cambio de proceso dentro de rutina de tratamiento
  - Algunos SO usan pila específica para int. SW sistema (1/procesador)

Uso de int. software de sistema

```
intX() {
    .......
    .......
}
intY() {
    .......
    activa_int_software_sistema();
    .......
}
tratamiento_int_software_sistema() {
    Realiza la operación diferida
}
```



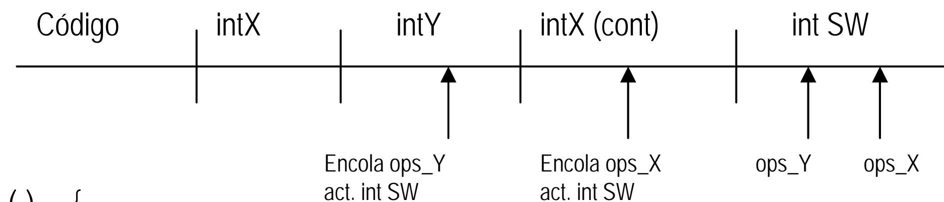
Implementación de int. software de sistema

```
intX() {
   .......
   rutina_fin_int();
}
intY() {
   .......
   activa_int_software_sistema();
   .......
   rutina_fin_int();
}
tratamiento_int_software_sistema() {Realiza la operación diferida}
```

```
activa_int_software_sistema () {
   int_SW_pendiente = true;
   ....... }
rutina_fin_int () {
   Si (int_SW_pendiente Y retorna a usuario
      O a evento síncrono){
        int_SW_pendiente = false;
        Habilitar interrupciones;
        tratamiento_int_software();}}
```

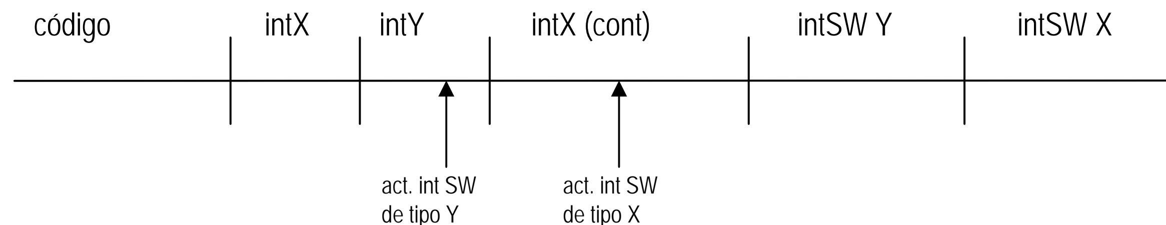
Ejemplo con int. SW sistema única

```
intX() {
   Operaciones críticas
   Encola operación (ops_X, datos_X)
   activa_int_software_sistema();
}
intY() {
   Operaciones críticas
   Encola operación (ops_Y, datos_Y)
   activa_int_software_sistema();
}
ops_X() {
   ........
}
ops_Y() {
   ........
}
tratamiento_int_software_sistema() {
   Mientras (haya operaciones diferidas) {
        Extraer operación → (func, datos)
        func(datos);
   }
}
```



Ejemplo con int. SW sistema múltiple

```
intX() {
   Operaciones críticas
   activa_int_software_sistema(DISPO_X);
}
intY() {
   Operaciones críticas
   activa_int_software_sistema(DISPO_Y);
}
tratamiento_int_SW_sist_X() {
   ........
}
tratamiento_int_SW_sist_Y() {
   ........
}
```



Interrupciones software de proceso

- Dirigida a un proceso: se trata cuando se cumplan condiciones y...
  - ese proceso esté en ejecución
- Implementación: información int SW de proceso pendiente en BCP
- Posibles aplicaciones:
  - Implementación de cambios de contexto involuntarios (planificación)
  - Terminación involuntaria de un proceso (abortar)
  - Implementación de operaciones asíncronas
- Pueden ser expulsivas o no expulsivas
  - Si eventos síncronos anidados, ¿esperan a que se completen?
  - Como veremos:
    - Int. SW de proceso para abortar no expulsiva
    - Int. SW de proceso para planificación: expulsiva o no expulsiva

Interrupciones software de proceso

- Presente de una forma u otra en todos los SSOO
  - Windows: ints. SW de proceso *Dispatch* y APC
  - Linux no usa este término pero lo implementa implícitamente
- En multiprocesador, si proceso destino int SW ejecuta en otra CPU
  - Enviar también IPI a procesador donde ejecuta
- Int. SW sistema mayor prioridad que int. SW proceso (DPC>APC)
  - Con frecuencia: rutina int. SW de sistema activa int. SW de proceso
- Ejecuta en contexto de un determinado proceso
  - Puede acceder a su mapa
  - Puede cambiar de proceso
  - Si pila de sistema e interrupciones, usa pila de sistema del proceso

Vida de un proceso

- Arranca en modo sistema pero “trucado” para pasar a usuario
- Ejecuta un programa, en modo usuario excepto mientras:
  - Procesa una llamada al sistema que ha invocado
  - Trata una excepción que ha generado
  - Trata una interrupción que se produce mientras ejecuta
- Termina ejecución, voluntaria o involuntaria, en modo sistema
- Multiplexación de procesos:
  - Durante el tratamiento de un evento el proceso cede el procesador

Procesos/hilos de núcleo

- Proceso (o hilo) de núcleo
- Ejecuta sólo código del SO, siempre en modo sistema
  - No tienen mapa de memoria de usuario asociado
  - Toma prestado el del último proc. de usuario que ejecutó en esa UCP
  - Cambio a p. núcleo eficiente: no requiere cambio de mapa memoria
- Normalmente se crean en la fase inicial del SO
  - Módulos del SO pueden crear más procesos de núcleo
- Realiza labores en el contexto de un proceso independiente
  - Pueden realizar operaciones de bloqueo
  - Pero no acceder a direcciones de memoria de usuario
- Normalmente, alta prioridad, pero no siempre (p.e. proceso nulo)
- Colas predefinidas de trabajos servidas por procesos de núcleo
  - En vez de crear un nuevo proceso de núcleo, se encola trabajo

Procesos del sistema

- No confundirlos con "procesos de núcleo"
- Procesos de usuario creados por el *superusuario*
- Ejecutan en modo usuario
  - "Pero sus llamadas al sistema son siempre atendidas"
- En sistemas monolíticos realizan labores del sistema
  - como *spooling* o servicios de red (demonios de UNIX)
- En sistemas micronúcleo realizan funcionalidades clásicas del SO
  - como la gestión de ficheros

Recapitulación de la gestión de eventos

- Actividades del SO encuadradas en alguna de estas categorías:
  - llamadas, excepciones, int HW y SW, procesos de núcleo
- Diseñador de módulo del SO, ¿cómo decide a qué contexto asociar las distintas funcionalidades del módulo?
- Criterios para asignar una acción a un contexto de ejecución:
  - Si vinculada con evento síncrono → incluir en rutina del evento
  - Si vinculada con evento asíncrono:
    - si acción crítica o urgente → incluir en rutina de interrupción
    - si no urgente y no requiere bloqueos → en rutina int. SW sistema
    - si requiere bloqueos → dentro de un proceso de núcleo
      - O se crea un nuevo proceso de núcleo
      - O se inserta en cola predefinida
  - Mejor si se puede hacer en modo usuario

Preguntas de repaso (1)

	Referencia dir. usuario	Cambio de proceso
Llamada al sistema		
Excepción (p.e. fallo página)		
Interrupciones de dispositivo		
Interrupciones SW de sistema		
Interrupciones SW de proceso		
Proceso/*thread de núcleo*		

Preguntas de repaso (2)

	Referencia dir. usuario	Cambio de proceso
Llamada al sistema	SÍ	SÍ
Excepción (p.e. fallo página)	SÍ	SÍ
Interrupciones de dispositivo	NO	NO
Interrupciones SW de sistema	NO	NO
Interrupciones SW de proceso	SÍ	SÍ
Proceso/*thread de núcleo*	NO	SÍ

Índice

- Introducción
- Gestión interna de eventos
- Implementación del modelo de procesos
  - Contexto de un proceso
  - Cambio de contexto
- Operaciones sobre los procesos
- Sincronización
- Implementación de hilos

Multiplexación de procesos

- Cambio de modo:
  - De usuario a sistema (excepción, llamada o interrupción)
  - De sistema a usuario (RETI)
- No hay c. de contexto: Mismo proceso en distinto modo
- Multiprogramación implica reasignar el procesador cuando:
  - Proceso se bloquea
  - Es conveniente ejecutar otro proceso en vez del actual
- Durante tratamiento de evento el proceso cede el procesador
- Necesidad salvar/restaurar información de los procesos
  - Info. del proceso → Contexto del proceso

Contexto de un proceso

- Información asociada con el proceso
- Almacenada en *Bloque de Control del Proceso* (BCP)
- Tabla de procesos: conjunto de estructuras de BCP
- Diseño de la tabla de procesos:
  - Vector de estructuras: mal aprovechamiento de la memoria
  - Vector de punteros a estructuras: Tamaño fijo
  - Lista de estructuras: se pierde capacidad de indexar
    - Uso de tablas *hash*

Tabla de procesos: vector de estructuras

BCP P3
Libre
BCP P8
BCP P1
Libre
BCP P4
BCP P2
…………

Tabla de procesos: vector de punteros

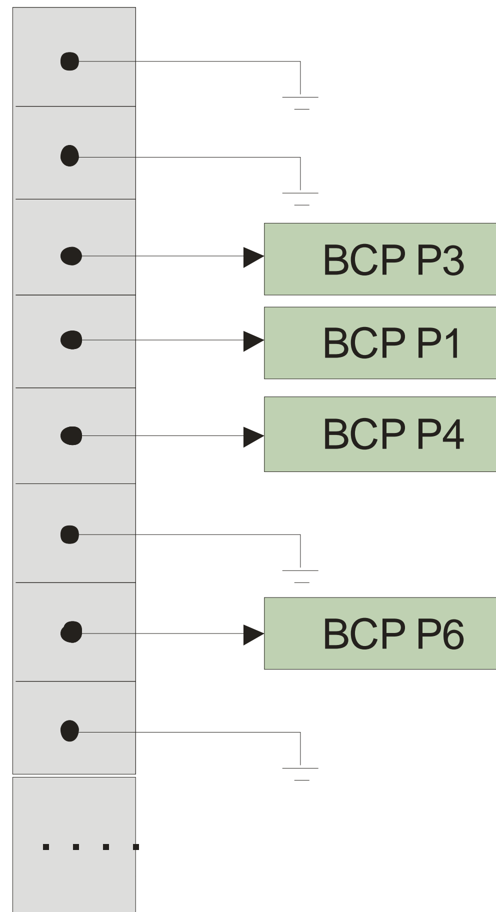
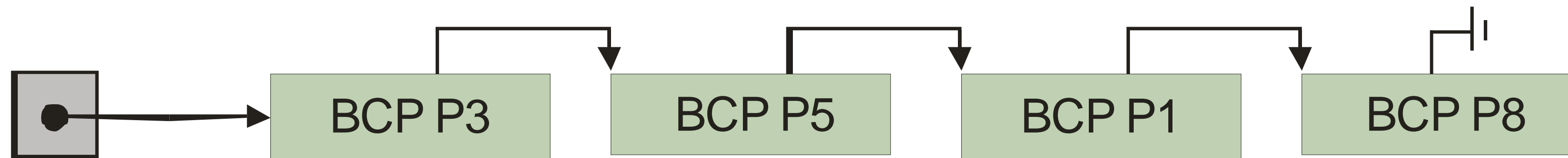


Tabla de procesos: lista de estructura



Bloque de Control del Proceso

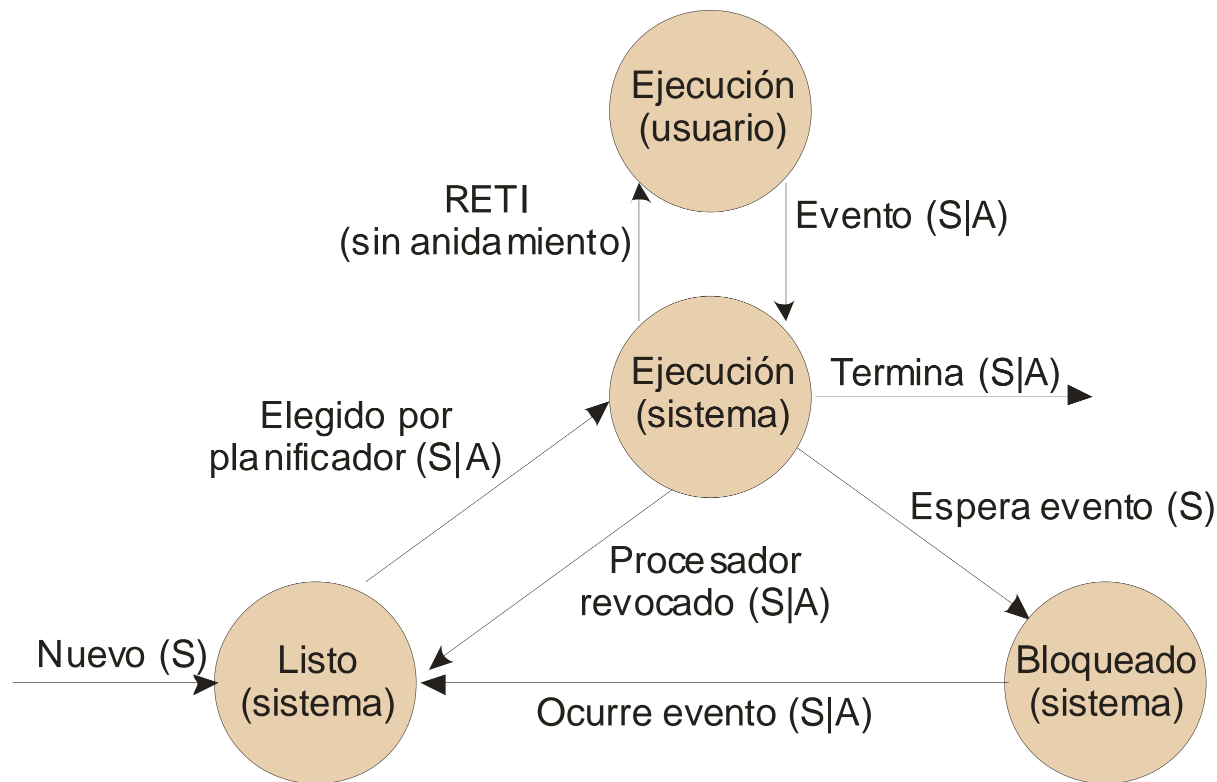
- Estado del proceso
- Copia del contenido de los registros del procesador
- Información de planificación (p.e. prioridad)
- Información sobre el mapa de memoria del proceso
  - Dirección de pila del sistema
- Información de ficheros, E/S, comunicación.
- Información de contabilidad y límites en el uso de recursos
- "Dueño" del proceso (uid y gid en UNIX)
- Identificador del proceso
- Punteros para construir listas de BCP
  - de procesos existentes, de procesos en mismo estado, parentesco, ...
- Consideraciones:
  - Si hilos: BCP incluye lista de hilos
  - Si micronúcleo: BCP repartido entre componentes

Estado de un proceso

- Elemento del contexto del proceso
  - En sistemas con hilos se aplica a cada hilo
- Durante su vida el proceso pasa por varios estados:
  - *En ejecución*: UCP asignada (usuario o sistema)
    - Núm. Procesos en ejecución = Núm. Procesadores
  - *Listo para ejecutar*: No hay procesador disponible para él
  - *Bloqueado*: Esperando un evento (un estado por cada evento)
- En sistemas con suspensión de procesos, estados adicionales:
  - *Suspendido y listo*: Expulsado pero listo para ejecutar
  - *Suspendido y bloqueado*: Expulsado y esperando un evento
- Se cumplen las siguientes propiedades:
  - Proceso bloqueado o listo se ha quedado en modo sistema
  - Proceso en ejecución puede estar en modo usuario o sistema
    - Para cambiar de estado debe estar en modo sistema

Transiciones entre estados

- Transición disparada siempre por activación del SO
  - No toda activación del SO implica c. de estado de un proceso.



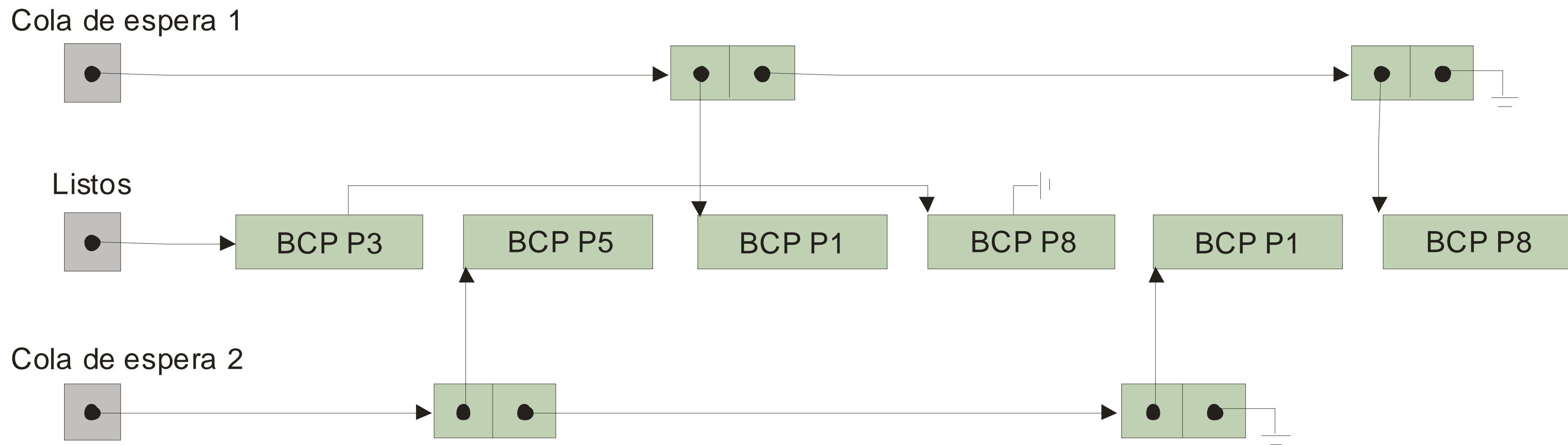
Estados de proceso en UNIX

- Peculiaridades respecto a diagrama convencional:
  - Estado *zombie*: hijo terminado pero padre no ha hecho *wait*
  - Estado *stopped*: parado por señal SIGSTOP
  - Estado de bloqueo dividido en:
    - espera interrumpible: señal saca de bloqueo a proceso
      - Para eventos cuya ocurrencia es impredecible (p. ej. int. teclado)
    - espera no interrumpible: sólo evento saca de bloqueo a proceso
      - Para eventos cuya ocurrencia es predecible (p. ej. int. disco)
  - En BCP no distingue entre estado listo y en ejecución

Colas de procesos

- SO enlaza BCPs en colas (BCP contiene punteros para ello)
  - Procesos en el mismo estado
- Cola de procesos listos
  - en muchos sistemas, proceso en ejecución también está en ella
- Cola de procesos bloqueados. Alternativas de diseño:
  - Una para todos los bloqueados: Ineficiente
  - Una por evento: gasto de memoria tolerable
  - Varios eventos por cola
- Si proceso en 1 cola en cada instante: basta con puntero en BCP
- En Linux, colas de bloqueados: *wait queues*
  - No usan puntero del BCP, incluyen el puntero al siguiente
- Cambio de estado implica pasar el BCP de una cola a otra

Colas de procesos en Linux



Cambio de contexto

- Cambio del proceso asignado al procesador
- Activación de SO que cambia estado de 2 procesos:
  - Proceso P pasa de *en ejecución* a *listo* o *bloqueado*
  - Proceso Q pasa de *listo* a *en ejecución*
  - Truco: activación del SO en la que proceso entrante != saliente
- C. contexto: Cambio de proceso
  - Estando el proceso (P) en modo sistema, el propio proceso:
    - cambia su estado a listo o bloqueado
    - reajusta BCP en colas de procesos
    - activa el planificador → selecciona Q
    - salva su contexto en BCP
      - PC "trucado" para que al volver a ejecutar se salte restauración
    - restaura contexto de Q del BCP → ya está ejecutando Q
      - incluye activar mapa de memoria de Q (operación costosa)

Aclaraciones sobre el cambio de contexto

- P no ejecutará hasta que otro proceso haga c. contexto a P
  - Visión externa: Procesos concurrentes
  - Visión interna: corrutinas dentro del sistema operativo
    - Cada ejecución de P va desde un c. contexto a otro.
  - Si P bloqueado antes debe desbloquearse
- Cuando proceso P vuelva a ejecutar:
  - Seguirá donde estaba (en el c. contexto), en modo sistema
- Dificultad para entender el código de gestión de procesos
- No toda activación del SO implica c. contexto
- No es "trabajo útil". Duración típica: decenas de microsegundos
- Código del c. contexto depende de arquitectura:
  - Escrito en ensamblador

Ejemplo aclaratorio

```
BCP *ant, *post;
void cambio (BCP *prev, BCP *sig) {
      printf("prev %d sig %d \n", prev->pid, sig->pid);
      ant = prev;   post = sig;
      cambio_contexto(&prev->contexto, &sig->contexto);
      printf("prev %d sig %d ant %d post %d\n",
                    prev->pid, sig->pid, ant->pid, post->pid); }
```

- Ejecución cíclica de P1, P2 y P3; Salida generada:
  - P1 reanuda su ejecución: prev 1 sig 2 ant 3 post 1
  - P1 invoca cambio(P1, P2): prev 1 sig 2
  - P2 reanuda su ejecución: prev 2 sig 3 ant 1 post 2
  - P2 invoca cambio(P2, P3): prev 2 sig 3
  - P3 reanuda su ejecución: prev 3 sig 1 ant 2 post 3
  - **P3** invoca cambio(P3, P1): prev 3 sig 1

Tipos de c. de contexto

- Cambio de contexto "voluntario" (C.C.V):
  - Llamada al sistema (o fallo de página) que espera por evento
  - Transición de *en ejecución* a *bloqueado*
  - Ej.: leer del terminal, bajar un semáforo cerrado, fallo de página
  - Motivo: Eficiencia en el uso del procesador
- Cambio de contexto "involuntario" (C.C.I):
  - SO le quita la UCP al proceso
  - Transición de *en ejecución* a *listo*
  - Ej.: fin de rodaja o pasa a *listo* proceso de mayor prioridad
  - Motivo: Reparto del procesador

C. contexto voluntario

- Ejemplo pseudo-código de leer el terminal:

```
leer() {
        ................................
        Si no hay datos disponibles
                proc_anterior = proc_actual;
                proc_anterior-> estado= BLOQUEADO;
                Mover proc_anterior de listos a lista del terminal;
                proc_actual = planificador();
                cambio_contexto(proc_anterior, proc_actual);
        ................................
}
```

Función de bloqueo

- Función que encapsula el C.C.V.
  - No debe invocarse desde rutina de interrupción
  - Proceso actual no está relacionado con la interrupción
- Prohibir int.: rutinas de int. también manipulan lista de listos

```
Bloquear (cola de bloqueo) {
        p_proc_actual->estado=BLOQUEADO;
        Prohibir interrupciones;
        Mover BCP de proceso actual a cola de bloqueo;
        Restaurar estado de interrupciones previo;
        cambio de contexto a proceso elegido por planificador;
}
```

C. contexto voluntario en llamada

- Proceso en modo usuario invoca llamada con posible bloqueo

```
sis_llamada_X () {
          . . . . . . . . . .
          Si (condición_1) Bloquear(cola de evento 1);
          . . . . . . . . . . ← Cuando se desbloquee seguirá por aquí

          Mientras ( ! condición_2) {
                    Bloquear(cola de evento 2);
                    . . . . . . . . . .  ← seguirá por aquí
          }
}
```

- Una llamada puede implicar varios bloqueos (fases)

C. contexto voluntario en fallo de página

- Ejemplo: Proceso provoca fallo de página:

```
fallo_página () {
        . . . . . . . . . .
        Si (no hay marco libre y página expulsada modificada) {
                Programar escritura en disco;
                Bloquear(cola de espera del disco);
                . . . . . . . . . . ← seguirá por aquí
        }
        // Traer nueva página
        Programar lectura de disco;
        Bloquear(cola de espera del disco);
        . . . . . . . . . .   ← seguirá por aquí
}
```

Desbloqueo de un proceso

- Cuando se cumpla el evento se produce el desbloqueo
  - No implica cambio de contexto
- Dentro de llamada o rutina de int. que causa desbloqueo

Si (condición) Desbloquear(cola de evento);

- Función *Desbloquear*
  - invocada desde rutina de int. o desde llamada

```
Desbloquear (cola de bloqueo) {
    Seleccionar proceso en la cola (primero si política FIFO);
    Estado de proceso elegido  = LISTO;
    Prohibir interrupciones;
    Mover BCP de elegido de cola del evento a cola de listos;
    Restaurar estado de interrupciones previo; }
```

C. de contexto involuntario

- Rutina de interrupción o llamada puede
  - desbloquear proceso más importante que alguno en ejecución
  - indicar que proceso actual ha consumido su turno
- Proceso a expulsar puede ser el que trató evento de expulsión o no
  - En monoprocesador, siempre será mismo proceso
- Proceso a expulsar involucrado en anidamiento de eventos del SO
  - Hay que esperar a que terminen la rutinas de tratamiento
- Necesidad de diferir c. contexto hasta que terminen
  - Se activa int SW de proceso dirigida a proceso a expulsar
  - Al terminar rutinas anidadas se trata int. SW que realiza c. contexto
- ¿Qué ocurre si activa una llamada al sistema y/o fallo de página?
  - Alternativas: Dejar también que terminen o no
    - Núcleos no expulsivos o expulsivos
      - Uso de int SW de proceso no expulsiva o expulsiva, respectivamente

Escenarios de c. c. involuntario

- Por int. de reloj que indica final de rodaja de p. actual

```
int_reloj () { ...
        Si (--pactual->rodaja == 0)
                activar int SW de planificación dirigida a pr. actual;
        .........
```

- Desbloqueo de proceso más prioritario

```
rutina_X () { ...
        Si (condición) {
                Desbloquear(cola de evento);
                Si  prio. desbloqueado > algún proc. en ejecución
                        activa int SW planif dirigida a pr. a expulsar;
        }  .........
```

- CCI en la rutina de tratamiento

```
tratamiento_int_SW_planificacion() {
        Llamar al planificador;
        Realizar CC involuntario al proceso elegido; }
```

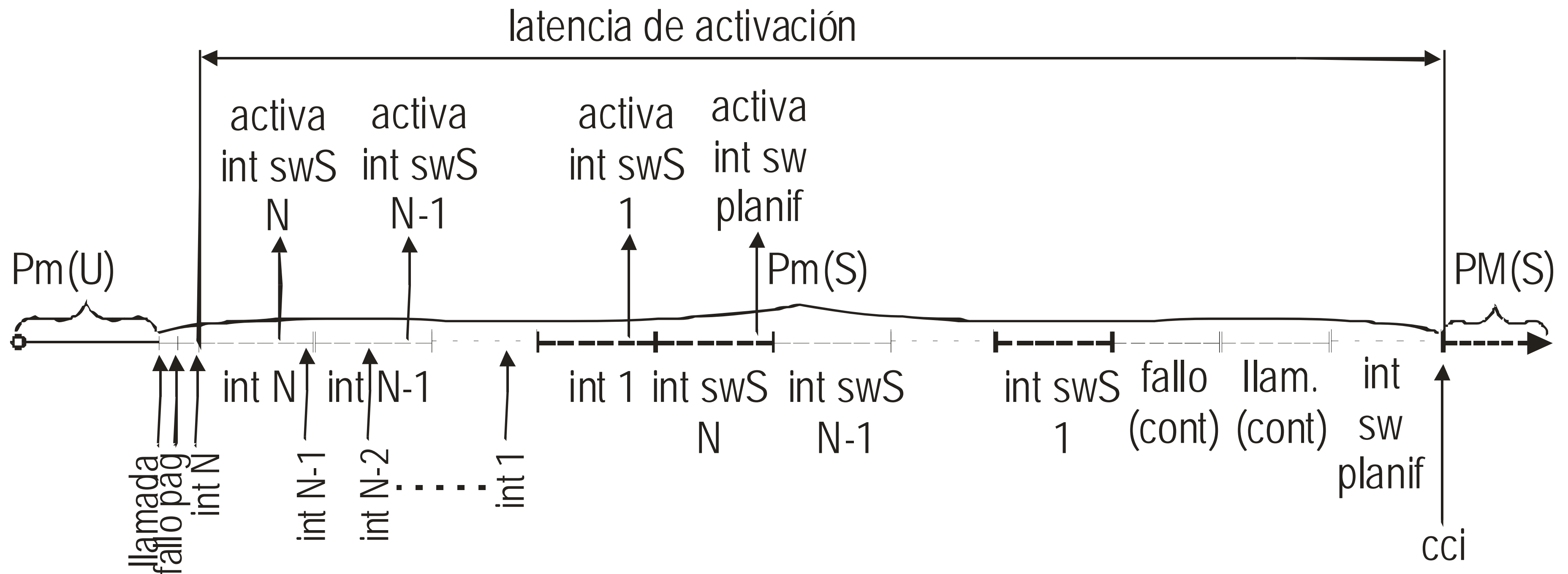
Núcleo no expulsivo

- C. contexto se difiere hasta que termina llamada y/o excepción
- Proceso en llamada continúa hasta que la termina o se bloquea
  - Mientras se procesan interrupciones pero no se realiza c. contexto
- Se reducen problemas de sincronización: ↑ fiable y ↓ sobrecarga
- Se limita concurrencia: no llamadas al sistema concurrentes
- Tiempo de respuesta alto (**latencia de activación** de proceso):
  - Proceso urgente desbloqueado por int. debe esperar en **peor caso**
    - latencia int. + fin fase llamada + fallo(s) pág. + ints. HW|SW + CC
  - Incluso si ejecuta proceso de prio. mínima (**inversión prioridades**)
  - Cuidado con fases de llamadas al sistema largas
- Se requiere interrupción software de proceso no expulsiva
- Llamada interrumpida continúa aunque haya interrupción SW
  - Si no termina la llamada porque se bloquea (CCV)
    - Desactivar int. SW de planificación para evitar replanificar

Niveles de interrupción en n. no expulsivo

Int HW N
....
Int HW 1
Int SW Sis M
....
Int SW Sis 1
Int SW Proc | Llam | Excep.

Latencia de activación n. no expulsivo



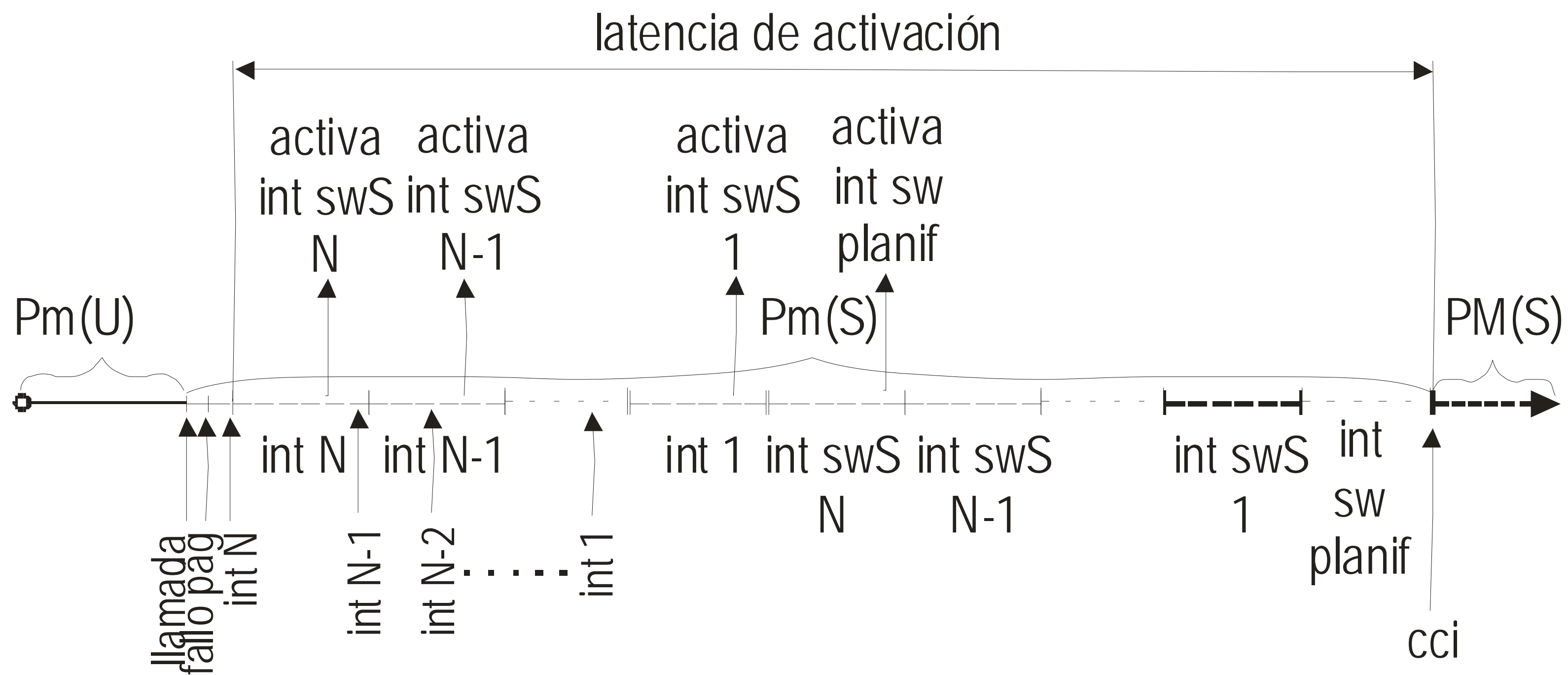
Núcleo expulsivo

- C. contexto se difiere sólo hasta fin de interrupciones
  - Interrupción SW expulsiva
  - Usado a partir de Linux 2.6 y de Windows 2000
- Permite llamadas concurrentes
- Problemas de sincronización: Necesidad de esquemas complejos
  - SO puede ser menos fiable y menos eficiente
- Mejora tiempo de respuesta/latencia medio y máximo:
  - Proceso urgente desbloqueado por int. debe esperar en peor caso
    - lat. interrupción + ints. HW/SW + CC + t. int. SW planif. inhibida
  - Reduce inversión de prioridades
- Se requiere interrupción software de proceso expulsiva
- Menor eficiencia que núcleo expulsivo
  - Sobrecarga de sincronización
  - Mayor número de cambios de contexto

Niveles de interrupción en n. expulsivo

Int HW N
....
Int HW 1
Int SW Sis M
....
Int SW Sis 1
Int SW Plan
Int SW Term | Llam | Excep.

Latencia de activación n. expulsivo



Núcleo para tiempo real

- TR: Garantizar plazos requeridos por aplicación
- Reducir peor latencia de interrupción y de activación en sistema
- ¿Núcleo expulsivo con bajas latencias es suficiente?
- TR requiere modelo integrado de prioridad de procesos/interrup.
  - SO propósito general: interrupciones siempre más prioritarias
- ¿N. expulsivo que ajuste nivel int. a prioridad proceso es suficiente?
  - No, ¿Cómo se completa rutina int. inicial en este escenario?
    - Int. baja prioridad interrumpida por int. alta
    - Int. alta desbloquea proceso alta prioridad y activa su ejecución
- Posible solución: Int. HW y SW sistema como procesos de núcleo
  - Modelo integrado de prioridades + CC en rutina de interrupción
- Mejora tiempo de respuesta/latencia medio y máximo:
  - Proceso urgente desbloqueado por int. debe esperar en peor caso
    - latencia interrupción + 2 CC (no depende de interrupciones)

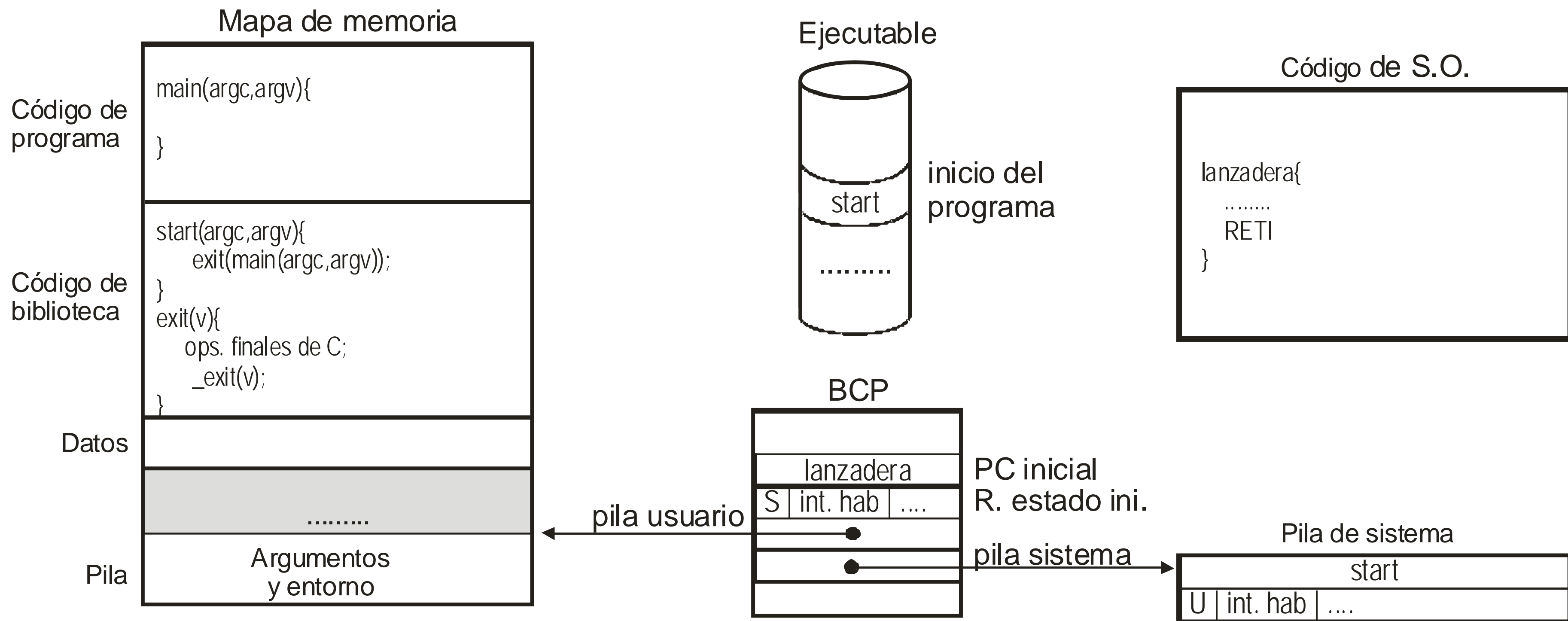
Índice

- Introducción
- Gestión interna de eventos
- Implementación del modelo de procesos
- Operaciones sobre los procesos
- Sincronización
- Implementación de hilos

Operaciones sobre procesos

- Vida de un proceso: falta nacimiento y defunción
- Creación de un proceso:
  - Convencional: Nuevo proceso ejecuta nuevo programa
  - UNIX: `fork` y `exec`
- Terminación de un proceso
  - Voluntaria
  - Involuntaria

Contexto inicial del proceso



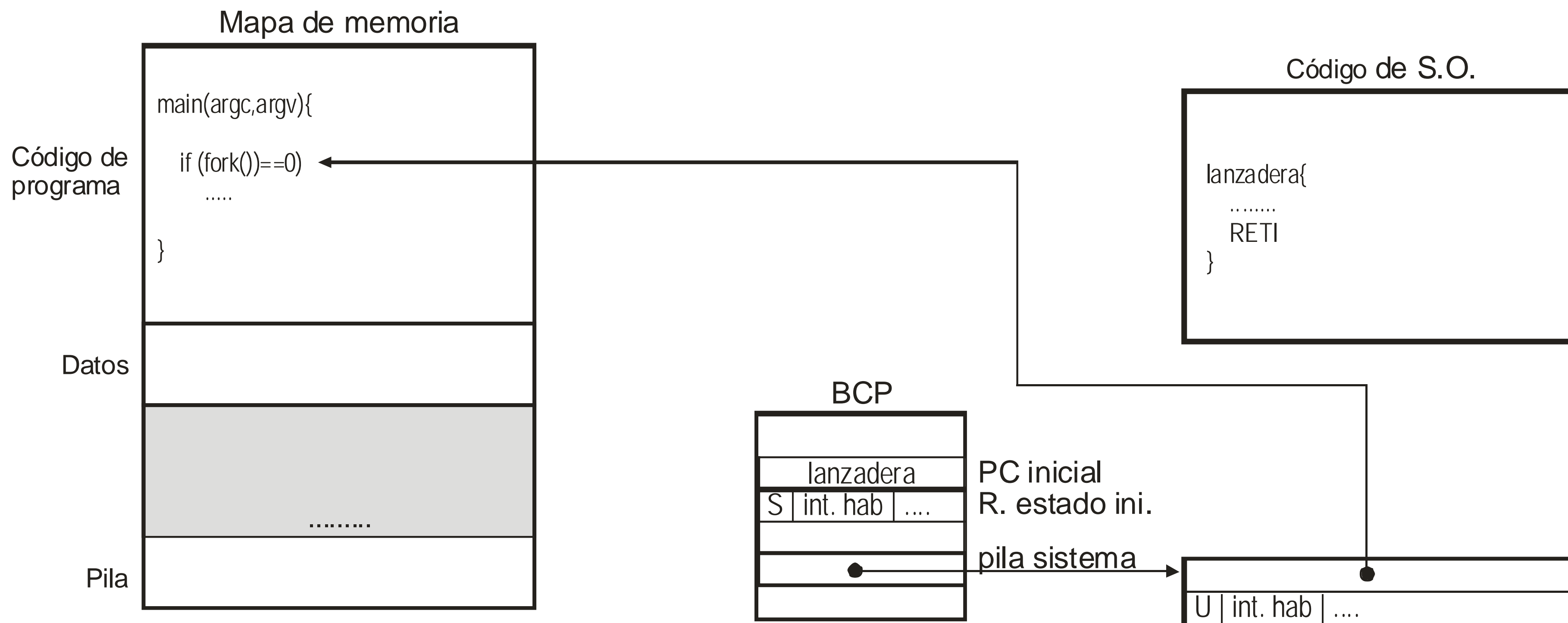
Creación de un proceso

- Operaciones típicas para modelo convencional
  - Reserva BCP
  - Leer fichero ejecutable
  - Crear mapa de memoria a partir del ejecutable
  - Crear pila inicial de usuario con entorno y args. del proceso
  - Crear pila del sistema con dir. del punto de arranque del programa
  - Iniciar BCP. Entre otras operaciones:
    - Crear contexto inicial en BCP
    - Estado = LISTO
  - Incluir BCP en cola de listos
- En UNIX estas operaciones están repartidas entre:
  - FORK
  - EXEC

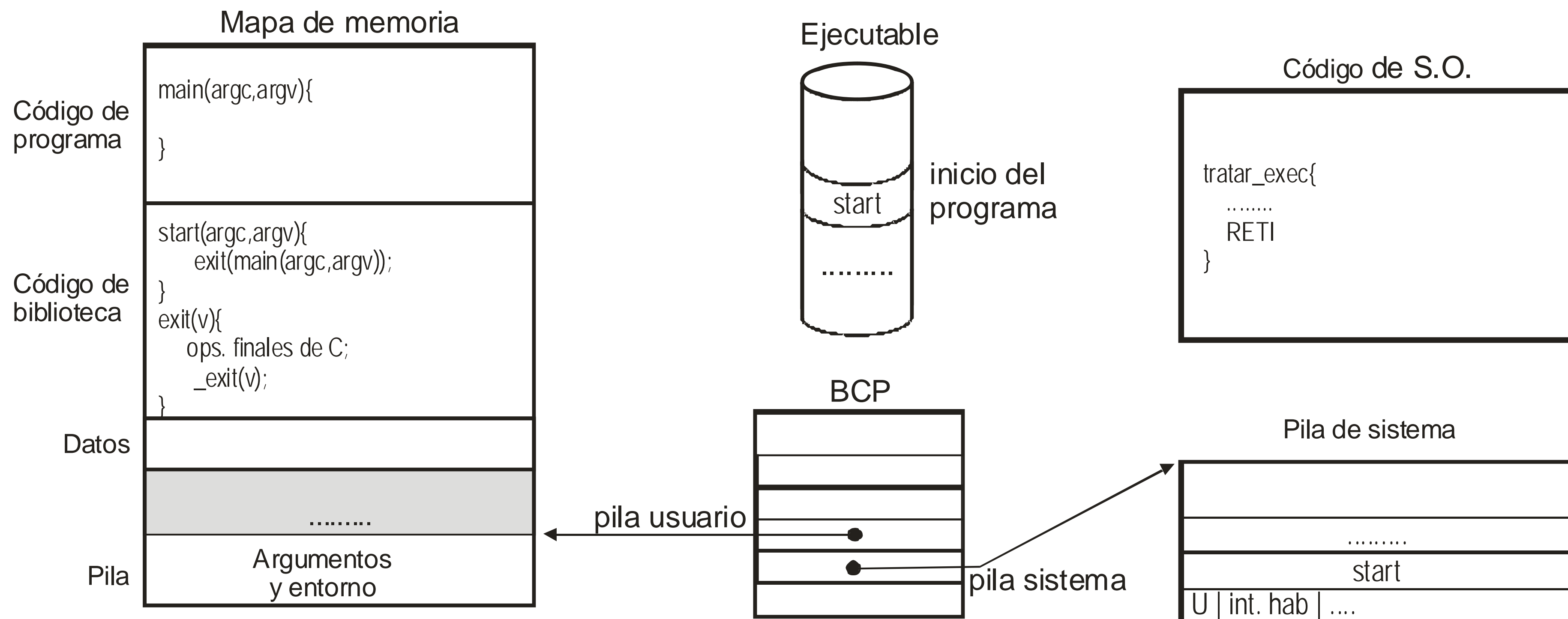
Creación de procesos en UNIX

- Operaciones en FORK
  - Reserva BCP
  - Copiar BCP del padre
  - Duplicar mapa de memoria del padre
  - Estado = LISTO + Incluir BCP en cola de listos
- Operaciones en EXEC
  - Leer fichero ejecutable
  - Liberar mapa de memoria del proceso
  - Crear nuevo mapa de memoria a partir del ejecutable
  - Crear pila inicial de usuario con el entorno y args. del proceso
  - Modificar algunos campos del BCP (p.e. señales capturadas)
  - Modifica pila de sistema para retornar a dir. de inicio del programa

Contexto del proceso (`fork`)



Contexto del proceso (exec)



Terminación de un proceso

- Tipo de terminación:
  - Voluntaria: invoca llamada al sistema (en UNIX, `_exit`, no `exit`)
  - Involuntaria:
    - Error de ejecución: excepciones (división por cero, ...)
    - Abortado: por un usuario (*Control-C*) u otro proceso (`kill`)
- UNIX, en vez de terminación involuntaria, se genera una señal:
  - ignorada: no hace nada
  - acción por defecto: terminar proceso
  - capturada: se manipula pila usuario para que ejecute func. captura
- ¿Qué ocurre con procesos hijos?
  - En UNIX pasan a depender del proceso `init`
- Influencia de la jerarquía:
  - UNIX: proceso terminado pasa a estado *Zombie*
  - Proceso no desaparece hasta que padre espera por él (`wait`)

Operaciones implicadas en la terminación

- Operaciones típicas (fin_proceso):
  - liberar mapa de memoria
  - cerrar ficheros y liberar otros recursos
  - eliminar BCP de cola de procesos listos
  - liberar BCP
  - liberar pila del sistema: ¿cómo y cuándo hacerlo?
    - El nuevo proceso, después del cambio de contexto
  - activa planificador y realiza c. de contexto al proceso elegido
- En UNIX estas operaciones están repartidas entre:
  - EXIT: realiza la mayor parte de las operaciones
  - WAIT: libera entrada de tabla de procesos y pila de sistema

Liberar pila de sistema (ejemplo previo)

```
BCP *ant, *post;
void cambio (BCP *prev, BCP *sig) {
    ant = prev;    post = sig;
    cambio_contexto(.....);
    if (??? -> estado == TERMINADO)
        liberar BCP y pila de sistema;
}
```

Liberar pila sistema (solución)

```
BCP *ant, *post;
void cambio (BCP *prev, BCP *sig) {
    ant = prev;    post = sig;
    cambio_contexto(.....);
    if (ant  -> estado == TERMINADO)
        liberar BCP y pila de sistema;
}
```

Escenarios de terminación (núcleo exp.)

- Estrategia global: proceso siempre se termina a sí mismo
  - Si proceso abortado en medio de llamada al sistema:
    - Dejar que se complete (¿y si no la completa?)
    - Abortarla pero dejando estado coherente
  - Uso de interrupción software de proceso no expulsiva
- P en ejecución, se produce evento y debe terminar
  - Evento es llamada o excepción: fin de proceso en su tratamiento
  - Evento es int. (p.e. Ctrl-C): Si en llam. al sistema, debe terminarla
    - Marcar proceso como terminado
    - Activar int. SW de proceso no expulsiva para P
    - Fin de proceso en su tratamiento
- Q aborta a P (en estado listo después de expulsión)
  - Igual que caso anterior
- Q aborta a P (en estado bloqueado o listo después de bloqueo)
  - Desbloquea a P, si estaba bloqueado, y lo marca como terminado
  - Cuando vuelva a ejecutar P, fin de proceso

Escenarios de terminación

```
llam_terminar{ .... fin_proceso(); ... }
tratar_exc_X{ .... fin_proceso(); ... }

tratar_int_Y{ // (p.e. Crtl-C) aborta proceso actual
    ....
    p_actual->estado=TERMINADO;
    activar_int_SW_terminacion(p_actual); ...
}
tratar_int_SW_terminacion() {
    ...
    if (p_actual->estado==TERMINADO) fin_proceso(); ...
}
```

Escenarios de terminación

```
tratar_kill() {  // aborta proceso P!=actual
    if (P->estado==BLOQUEADO) mover BCP de cola bloqueo a listos
    P->estado=TERMINADO;
    activar_int_SW_terminacion(P);... }
int bloquear() {
    if (p_actual->estado==TERMINADO) return -1; ...
    cambio_contexto(....);
    if (p_actual->estado==TERMINADO) return -1; ... }
leer_tuberia() {
    if (tuberia_vacia()) {
            ret=bloquear();
            if (ret==-1) {dejar estado coherente y fin_proceso()}
            ....
```

Recapitulación: “Vida” de un proceso

- Proceso P se inicia en modo sistema y pasa enseguida a usuario
- Cuando se produce un evento pasa a modo sistema y:
  - Si no implica c. contexto, vuelve a modo usuario
  - Si implica CCV (llamada o excepción de fallo de página)
    - P se queda en modo sistema en rutina c. c. de bloquear
    - cuando se desbloquea, pasa a listo pero sigue en mismo punto
    - cuando prosiga puede volver a bloquearse sin terminar llamada
  - Si implica CCI:
    - P se queda en m. sistema en rutina c. c. de int SW de planificación
    - cuando prosiga termina rutina int SW y continúa donde estaba
      - Si núcleo no expulsivo: ejecutando programa en modo usuario
      - Si expulsivo: ejecutando programa (U) o tratamiento e. síncrono
- Cuando termina voluntaria o involuntariamente vuelve a m. sistema

Preguntas de repaso (1)

	Núcleo no expulsivo	Núcleo expulsivo
Interrupción SW de sistema		
Interrupción SW de planificación		
Interrupción SW de terminación		

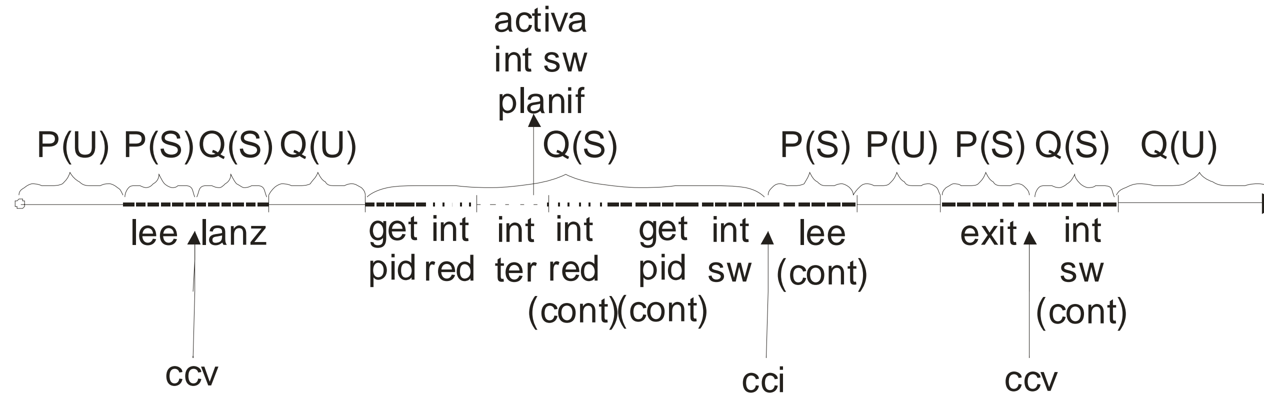
Preguntas de repaso (2)

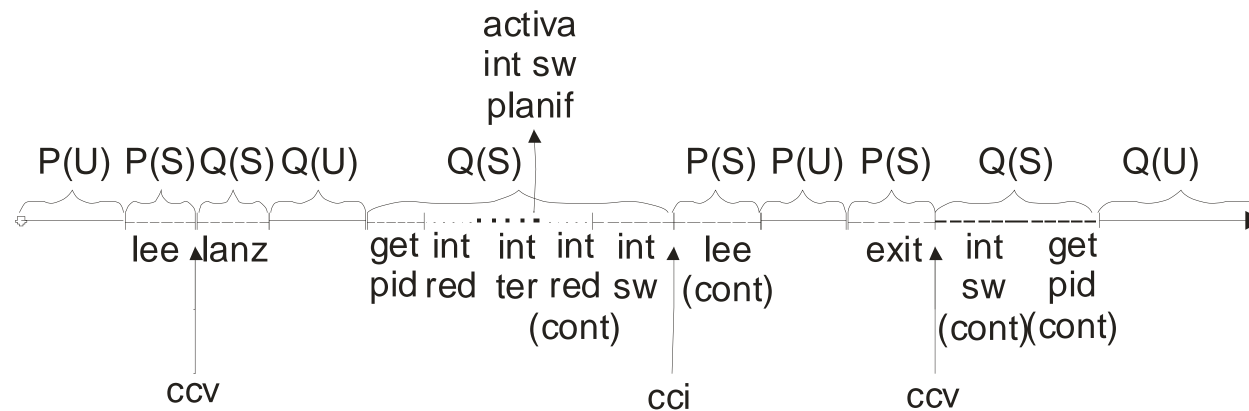
	Núcleo no expulsivo	Núcleo expulsivo
Interrupción SW de sistema	Expulsiva	Expulsiva
Interrupción SW de planificación	No expulsiva	Expulsiva
Interrupción SW de terminación	No expulsiva	No expulsiva

Ejemplo 1

- Prio(P) > Prio(Q).
- P en ejecución y Q es nuevo
- P lee del terminal con buffer vacío
- Q alterna fases de usuario con llamadas *getpid*
- En medio de *getpid*, int. de red, y justo a mitad, int. de terminal
  - Interrupciones de dispositivos no usan int. SW de sistema
- Después de leer, P *exit*

Ejemplo 1



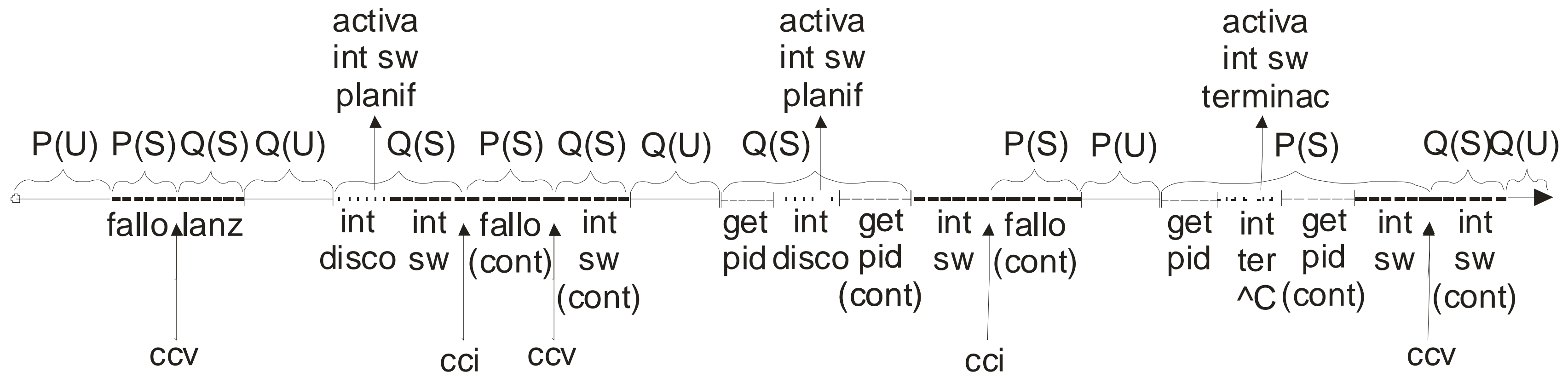


Ejemplo 2

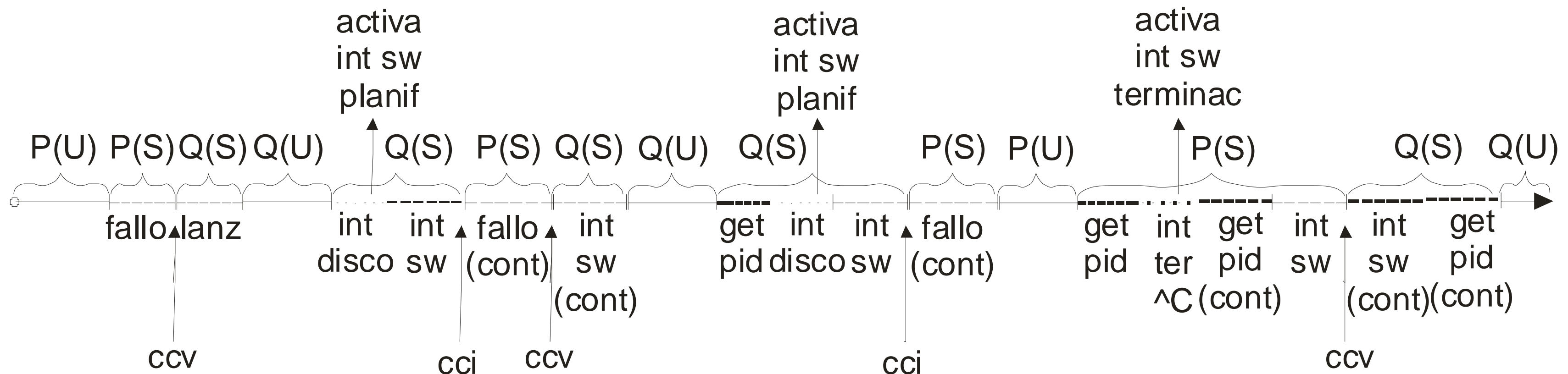
- Prio(P) > Prio(Q).
- P en ejecución y Q es nuevo
- P causa fallo de página
  - Requiere escribir página modificada a disco y leer de disco la nueva
- Q alterna fases de usuario con llamadas a *getpid*
- 1ª interrupción de disco ocurre con Q en modo usuario
  - Interrupción de disco no usa int. SW de sistema
- 2ª interrupción de disco con Q en medio de *getpid*
- Después de fallo de página, P llama a *getpid*
- En medio de llamada, usuario aborta proceso tecleando Ctrl-C
  - Interrupción de disco no usa int. SW de sistema

Ejemplo 2

versión no expulsiva



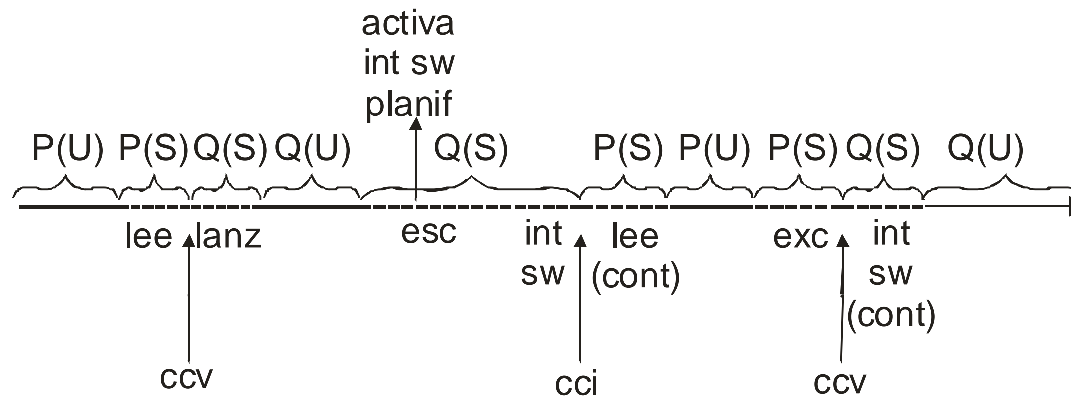
versión expulsiva

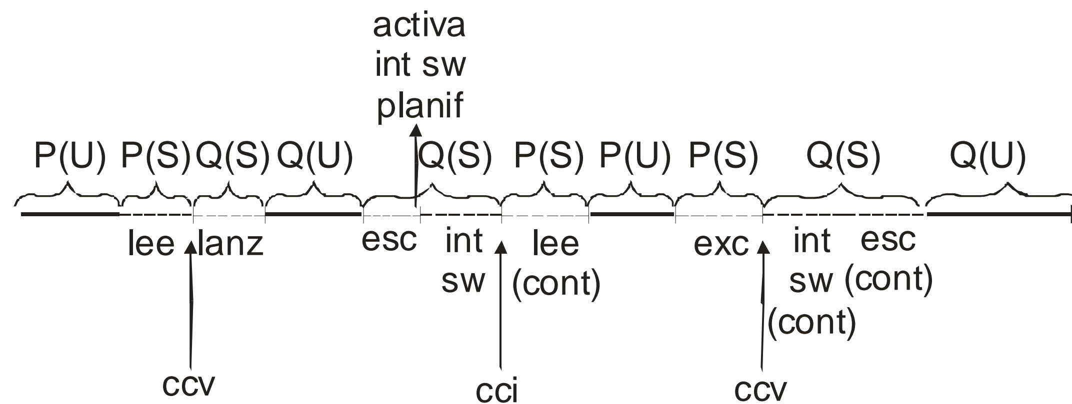


Ejemplo 3

- Prio(P) > Prio(Q)
- P en ejecución y Q es nuevo
- P lee de una tubería vacía
- Q escribe en la tubería
- Después de leer, P realiza una división por cero

Ejemplo 3



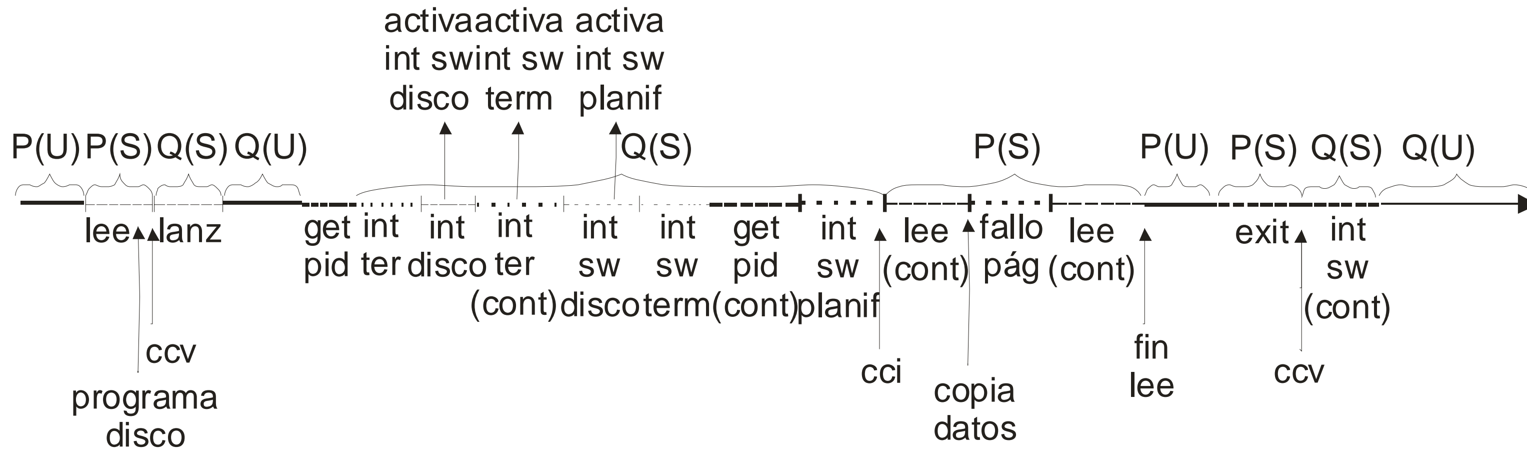


Ejemplo 4

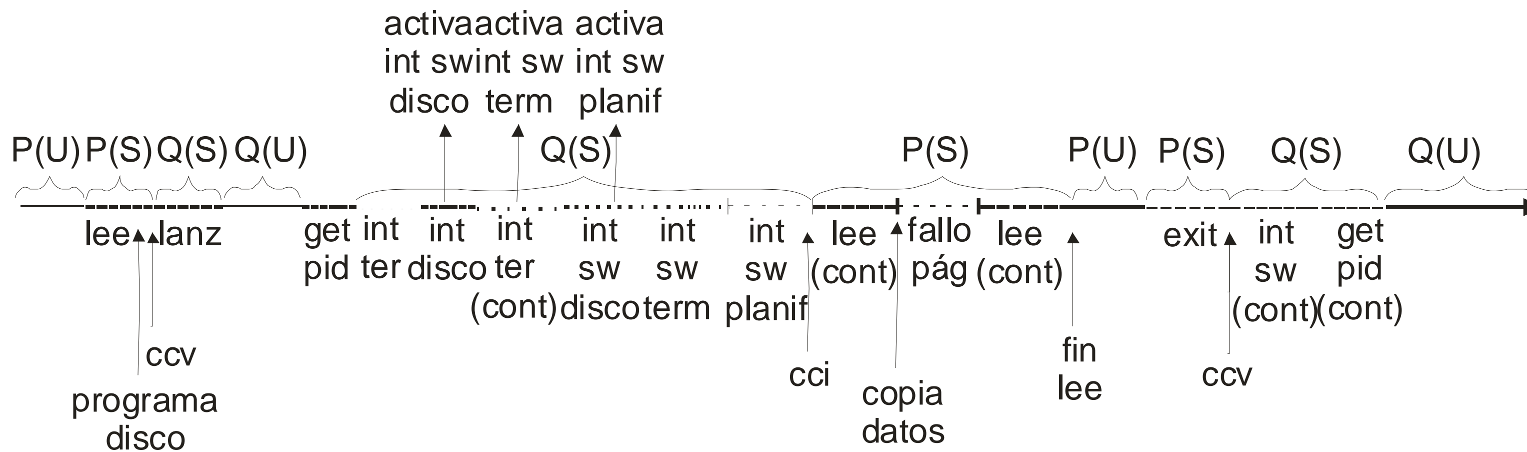
- Prio(P) > Prio(Q)
- P en ejecución y Q es nuevo
- P lee bloque de fichero sobre variable global sin valor inicial
  - Nunca accedida y ocupa sólo una página.
  - En el sistema hay marcos de página libres
  - Acceso a esa página → fallo que no requiere operación del disco
- Q ejecuta cálculos en modo usuario intercalados por llamada *getpid*
- Antes de int. disco, con Q en *getpid*, int. terminal de dato no pedido
- Después de leer, P *exit*
- Interrupciones de dispositivos usan int. SW de sistema

Ejemplo 4

versión no expulsiva



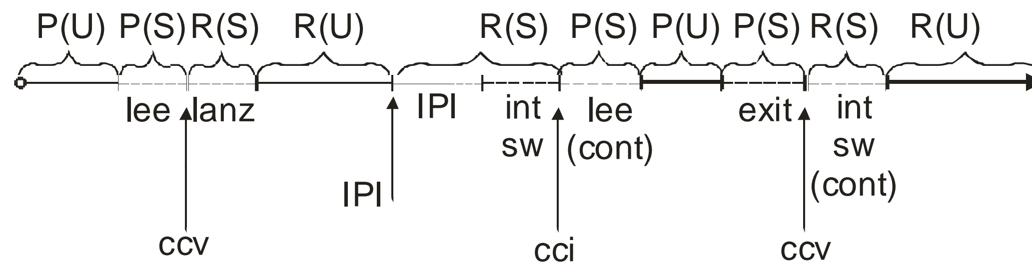
versión expulsiva

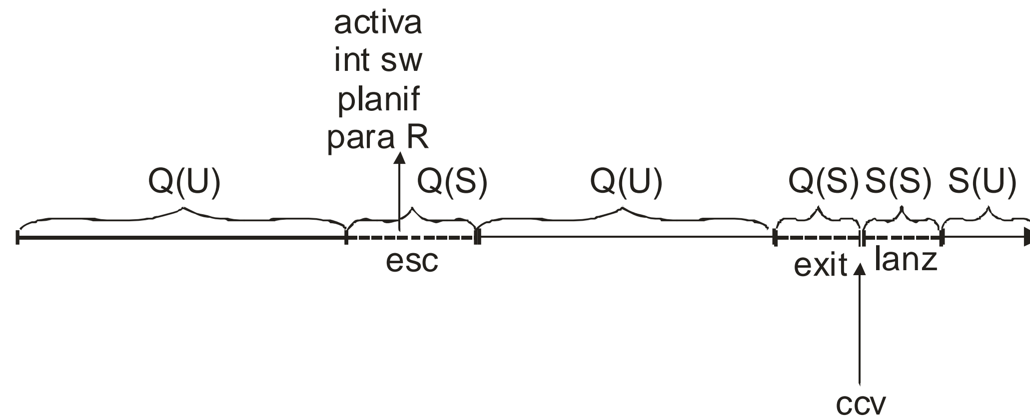


Ejemplo 5

- 2 procesadores
- Prio(P) > Prio(Q) > Prio(R) > Prio(S)
- P y Q en ejecución; R y S nuevos
- P lee de una tubería vacía
- Q escribe en la tubería
- R y S siempre en modo usuario
- Al terminar, P y Q *exit*
- Como no expulsión con proceso en modo sistema
  - Traza n. expulsivo = n. no expulsivo

Ejemplo 5





Índice

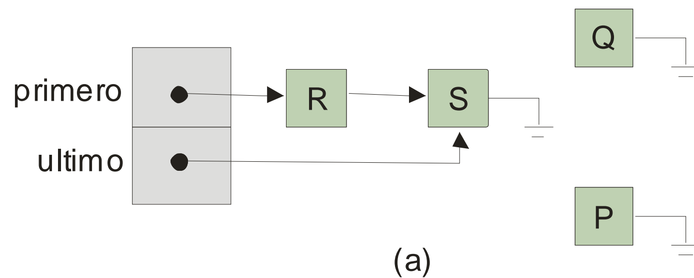
- Introducción
- Gestión interna de eventos
- Implementación del modelo de procesos
- Operaciones sobre los procesos
- Sincronización
  - Sincronización en el sistema operativo
  - Sincronización entre procesos de usuario
- Implementación de hilos

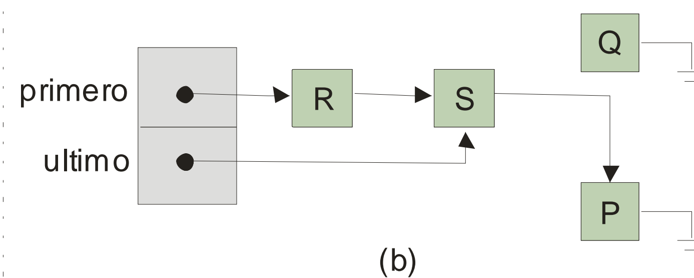
Problemas de sincronización en el SO

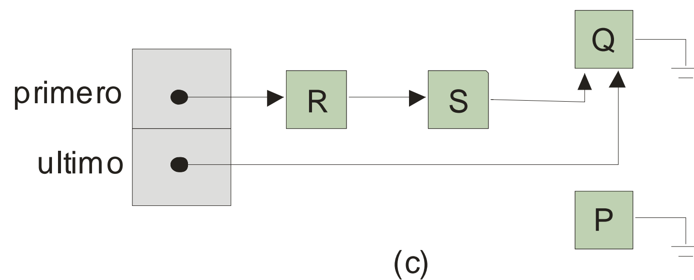
- SO es un programa con alto grado de concurrencia:
  - Sincronización compleja: difícil asegurar que funciona correctamente
  - Los problemas clásicos de sincronización provienen del SO
- Tipos de problemas de sincronización:
  1. Producidos por tratamiento de evento asíncrono
  2. Debidos a ejecución entremezclada de procesos
     - Por cambios de contexto involuntarios (2a) o voluntarios (2b)
  3. Específicos de multiprocesadores debidos al paralelismo real
  4. Específicos de s. de tiempo real: inversión de prioridades no acotada
- Es necesario crear secciones críticas dentro del SO
  - ¿Por qué no usar semáforos (o mutex, ...)?
    - Sí, pero, ¿cómo asegurar la atomicidad de los mismos?
    - Además, una rutina de interrupción no puede bloquearse

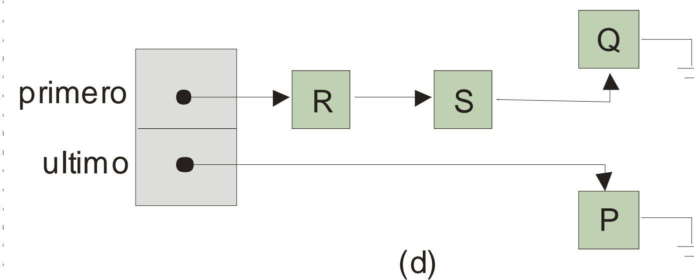
Ejemplo de problemas tipo 1

```
insertar_ultimo(lista, BCP){
    lista->ultimo->siguiente = BCP;
    lista->ultimo = BCP; }
```









Ejemplo de problemas tipo 2a

- Condición de carrera si ejecución concurrente de llamadas
  - O tratamiento de excepciones o procesos de núcleo
  - Ejemplo: Mismo BCP para varios procesos

```
crear_proceso(...) {
    ...........
    pos=BuscarBCPLibre();
    tabla_procesos[pos].ocupada = true;
    ...........
}
```

Ejemplo de problemas tipo 2b

- Condición de carrera incluso sin ejecución concurrente de llamadas
  - Ejemplo: operaciones sobre fichero que se pretende que sean atómicas

```
write(...) {
        Por cada bloque especificado
                Escribir bloque // operación que puede causar bloqueos
}
```

Sincronización en núcleo no expulsivo

- Sincronización entre llamada/rutina de int. y otra rutina de int.:
  - Se eleva nivel interrupción durante sección crítica
    - Si sistema sin niveles, se prohíben interrupciones
  - Si int. conflictiva es int. SW de sistema, se inhibe la misma
    - Int. SW de proceso no causan conflictos por no ser expulsivas
  - Minimizar zona afectada por inhibición reduce latencia de int. y act.
  - Mientras int. inhibidas no debería haber CCV (CCI no son posibles)
- Problemas por cambios de contexto involuntarios:
  - No hay: no se permiten llamadas concurrentes
- Problemas por cambios de contexto voluntarios:
  - Sección crítica que incluya bloqueos: uso de semáforos
  - Operaciones de semáforos son atómicas por ser no expulsivo
  - Ejemplo: lecturas y escrituras sobre fichero deben ser atómicas

Soluciones para núcleo no expulsivo

- Problemas de tipo 1:

```
insertar_ultimo(lista, BCP){
        nivel_anterior = fijar_nivel_int(NIVEL_MAXIMO);
        lista->ultimo->siguiente = BCP; lista->ultimo = BCP;
        fijar_nivel_int(nivel_anterior);  }
```

- Problemas de tipo 2b:

```
write(...) {
        bajar(inodo.sem)
        Por cada bloque especificado
                Escribir bloque // operación puede causar bloqueos
        subir(inodo.sem);  }
```

Resumen de conflictos para no expulsivo

Rutina a estudiar	Rutina conflictiva				
	Llamada	Excep	I. SW sis	I. dis. mn	I. dis. mx
Llamada	Semáf. si SC con blq	Semáf. si SC con blq	Inhibir int SW sis	Inhibir int. mín	Inhibir int. máx
Excepción	Semáf. si SC con blq	Semáf. si SC con blq	Inhibir int SW sis	Inhibir int. mín	Inhibir int. máx
Int. SW sistema	—	—	—	Inhibir int. mín	Inhibir int. máx
Int. dispo. mínima	—	—	—	—	Inhibir int. máx
Int. dispo. máxima	—	—	—	—	—

Sincronización en núcleo expulsivo

- Sincronización entre llamada/rutina de int. y otra rutina de int.:
  - Igual que no expulsivo: eleva nivel int. durante sección crítica
  - Mientras int. inhibidas no debería haber CCV ni CCI
- Sincronización entre llamadas concurrentes:
  - Impedir CCI durante región crítica: inhibir int. SW de planificación
  - Inadecuada si s. crítica larga (afecta lat. activación) o con bloqueo
  - Afecta también a procesos no involucrados en el conflicto
- Solución alternativa: Uso de semáforos
  - Prohibición de int. SW de planificación para atomicidad en semáforo
    - pero sección crítica con int. software habilitada
  - Sin embargo, si sección crítica muy corta
    - Más eficiente prohibir int. SW planificación que semáforos
  - Granularidad de los semáforos:
    - Paralelismo frente a sobrecarga

Soluciones para núcleo expulsivo

- Basada en impedir cambio de contexto involuntario:

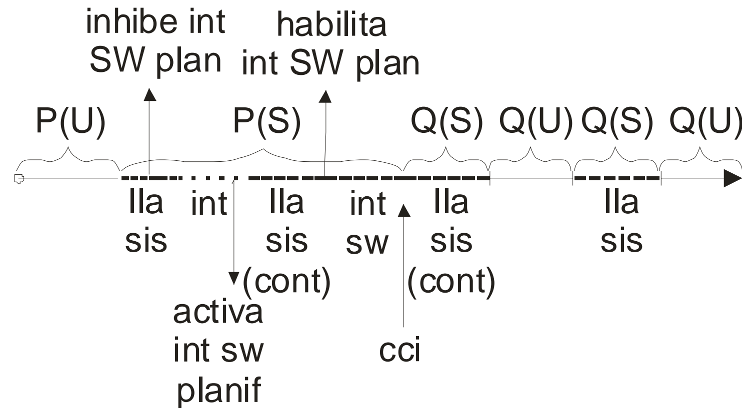
```
crear_proceso(...) {
        nivel_ant=fijar_nivel_int(NIVEL_INT_SW_PLANIFICACION);
        pos=BuscarBCPLibre();
        tabla_procesos[pos].libre = false;
        fijar_nivel_int(nivel_ant); }
```

- Basada en semáforos (si sección crítica larga y/o con bloqueo):

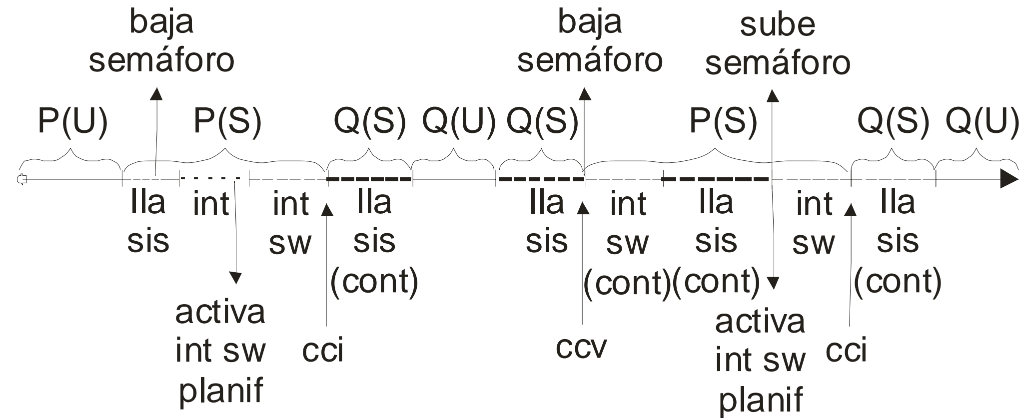
```
crear_proceso(...) {
        bajar(semaforo_tabla_procesos);
        pos=BuscarBCPLibre();
        tabla_procesos[pos].libre = false;
        subir(semaforo_tabla_procesos); }
```

Eficiencia inhibir expulsión vs. semáforos

versión que inhibe expulsión



versión con semáforos



Resumen de conflictos para expulsivo

Rutina a estudiar	Rutina conflictiva				
	Llamada	Excep	I. SW sis	I. dis. mn	I. dis. mx
Llamada	Semáf. o Inh. Expul	Semáf. o Inh. Expul	Inhibir int SW sis	Inhibir int. mín	Inhibir int. máx
Excepción	Semáf. o Inh. Expul	Semáf. o Inh. Expul	Inhibir int SW sis	Inhibir int. mín	Inhibir int. máx
Int. SW sistema	—	—	—	Inhibir int. mín	Inhibir int. máx
Int. dispo. mínima	—	—	—	—	Inhibir int. máx
Int. dispo. máxima	—	—	—	—	—

Sincronización en multiprocesadores

- Nuevas situaciones problemáticas
  - Una llamada puede ejecutar mientras lo hace una rutina de int.
- Soluciones de sincronización no válidas para multiprocesador
  - Prohibir interrupción no impide que ejecute rutina en otras UCPs
- Técnica básica: Uso de *spinlocks*
  - Espera activa sobre variable usando accesos atómicos a memoria

```
spin_lock(int cerrojo) { while (TestAndSet(cerrojo) == 1); }
spin_unlock(int cerrojo) { cerrojo = 0;}
```

- Cambios de contexto y *spinlocks*
  - Ni CCV ni CCI mientras se mantiene uno
  - Pero sí CCI mientras espera activa

Sincronización con interrupciones en MP

- Sincronización entre llamada/rutina de int. y otra rutina de int.:
  - Rutinas usan *spinlock* y rut. interrumpida inhibe localmente interr.

```
int spin_lock_interrupcion(int cerrojo, int nivel) {
   nivel_anterior = fijar_nivel_int(nivel);
   spin_lock(cerrojo);
   return nivel_anterior; }
spin_unlock_interrupcion(int cerrojo, int nivel) {
   spin_unlock(cerrojo);
   fijar_nivel_int(nivel); }

insertar_ultimo(lista, BCP){
   prev=spin_lock_interrupcion(lista->cerrojo,NIVEL_MAX);
   lista->ultimo->siguiente = BCP;
   lista->ultimo = BCP;
   spin_unlock_interrupcion(lista->cerrojo, prev); }
```

Sincronización entre llamadas en MP

- Sincronización entre llamadas concurrentes:
  - *spinlocks* + int SW de planificación inhibida si núcleo expulsivo

```
crear_proceso(...) {
        nivel_ant=spin_lock_interrupcion(cerrojo_tabla_proc,
                NIVEL_INT_SW_PLANIFICACION);
        pos=BuscarBCPLibre();
        tabla_procesos[pos].libre = false;
        spin_unlock_interrupcion(cerrojo_tabla_proc, nivel_ant); }
```

  - Granularidad de *spinlock*: paralelismo vs. sobrecarga
  - Si s. crítica larga o con bloqueo: semáforos
    - Atomicidad de semáforos gracias a *spinlocks*
    - Granularidad de semáforos

Resumen de conflictos para MP expulsivo

Rutina a estudiar	Rutina conflictiva				
	Llamada	Excep	I. SW sis	I. dis. mn	I. dis. mx
Llamada	Sem. o spin sin expul.	Sem. o spin sin expul.	Spin + Inh. int SW sis	Spin + Inh. int mín	Spin + Inh. int máx
Excepción	Sem. o spin sin expul.	Sem. o spin sin expul.	Spin + Inh. int SW sis	Spin + Inh. int mín	Spin + Inh. int máx
Int. SW sistema	Spin	Spin	Spin	Spin + Inh. int mín	Spin + Inh. int máx
Int. dispo. mínima	Spin	Spin	Spin	Spin	Spin + Inh. int máx
Int. dispo. máxima	Spin	Spin	Spin	Spin	Spin

Contextos atómicos

- Contexto atómico si se cumple **alguna** de estas condiciones:
  - Rutina de interrupción de un dispositivo
  - Rutina de interrupción software de sistema.
  - Prohibidas las interrupciones de los dispositivos.
  - Inhibidas las interrupciones software de sistema.
  - Deshabilitada la expulsión de procesos (no permitida int. SW planif.)
  - En posesión de un *spinlock*
- Sólo se puede hacer cambio de proceso
  - Si en contexto **no atómico**
- Sólo se puede acceder a mapa de usuario
  - Si en contexto **no atómico y no se trata de un proceso de núcleo**

Sincronización en núcleo de tiempo real

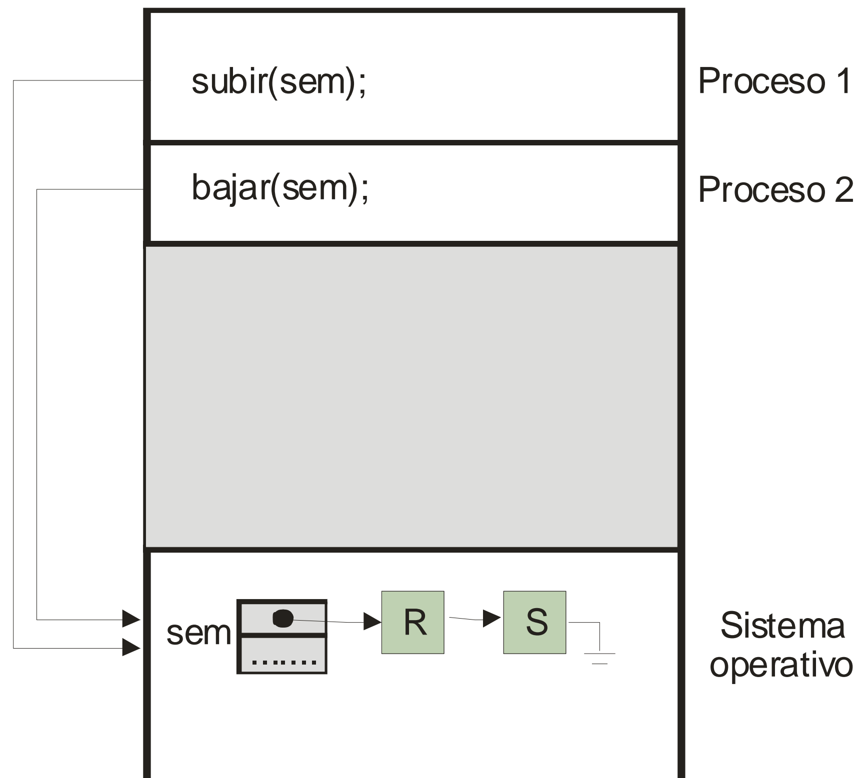
- Reto: controlar la inversión de prioridades
- Soluciones de SO propósito general no adecuadas en general:
  - Inhibir expulsiones: afecta a procesos no involucrados
  - Inhibir interrupciones: aumenta latencias
  - *Spinlocks*: procesador con *spinlock* no expulsiones
- Uso de semáforos para todo: afectan sólo a procesos involucrados
  - Para todo ya que int. HW y SW de sistema en procesos de núcleo
  - Sobrecarga en SC muy breves pero TR importan plazos no rapidez
  - Implementación de semáforos requiere *spin*, inhibir interrupciones, ...
    - Sí pero uso restringido y optimizado
- Semáforos deben respetar prioridades y controlar inversión de prio.
  - Uso de herencia de prioridades

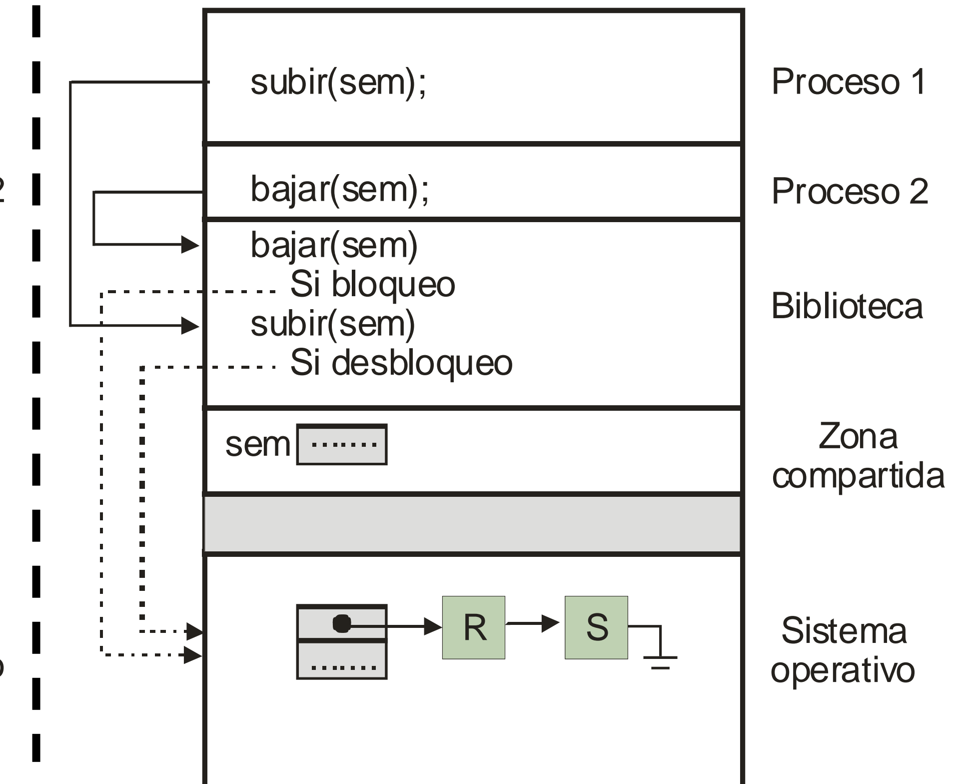
Herencia de prioridades

- Problema: inversión no acotada en uso de semáforos
  - P prioridad baja obtiene semáforo S
  - Q prioridad alta se bloquea al intentar obtener S
  - R prioridad media ejecuta: P no progresa → Tampoco Q
- Inversión de prioridades no acotada en el tiempo
- Solución: mientras P posee S y Q espera, Prio(Q) → Prio(P)
- Herencia de prioridades:
  - Prio(P) ← máx(prio_est(P),prio(proc esperando sem poseídos por P))
- Ajuste no trivial: debe propagarse prioridad de forma transitiva
  - P prioridad baja obtiene semáforo S1
  - Q prioridad media obtiene S2 y se bloquea al intentar obtener S1
    - Prio(Q) → Prio(P)
  - R prioridad alta se bloquea al intentar obtener S2
    - Prio(R) → Prio(Q) → Prio(P)

Sincronización de aplicaciones

- Mecanismos de sincronización de programas basados en internos
  - Aunque con funcionalidades añadidas
- Puede implementarlo sólo el SO o con parte de código de usuario





Índice

- Introducción
- Gestión interna de eventos
- Implementación del modelo de procesos
- Operaciones sobre los procesos
- Sincronización
- Implementación de hilos

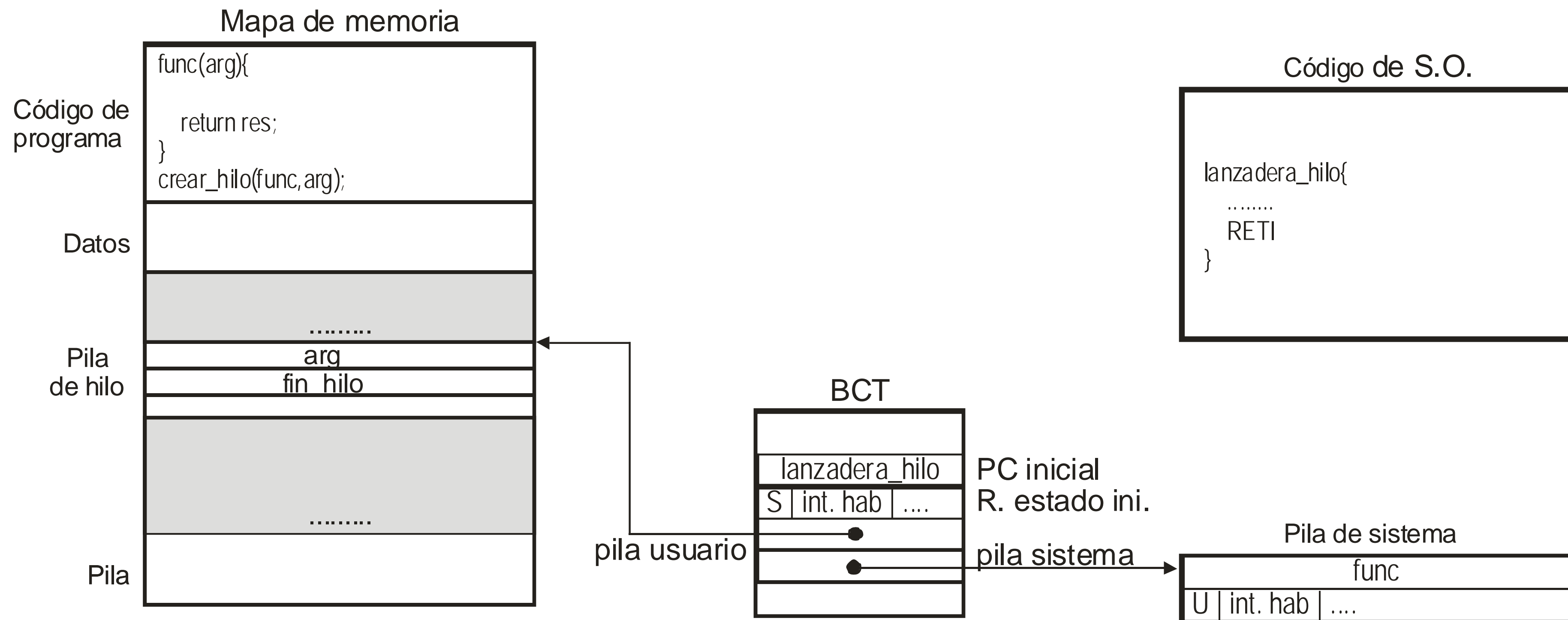
Hilos (*threads*)

- Concepto moderno de proceso: múltiples hilos de ejecución
  - Difícil incorporar a SO existente (¿`fork`, `exec`, señales con hilos?)
- El proceso se corresponde con un entorno de ejecución:
  - Un mapa de memoria
  - Un conjunto de recursos asociados (ficheros, semáforos, ...)
  - Un conjunto de hilos
- Proceso tiene un hilo implícito: el flujo de ejecución inicial
- Hilos de un mismo proceso comparten:
  - Mapa de memoria y recursos asociados al proceso
- Cada hilo tiene recursos propios:
  - Pila (de usuario y sistema), estado y copia de contenido de regs.
- Ventajas del uso de hilos vs. procesos convencionales:
  - Permiten expresar concurrencia “ligera” y fuertemente acoplada

Implementación de hilos

- BCP sólo contiene info. gral. del proceso + lista de hilos
- BCT (*Bloque de Control de Hilo*):
  - Info. específica del hilo
    - estado, copia de regs., pilas de usuario y sistema, prioridad, puntero a BCP del proceso, etc.
- Colas de listos y bloqueo contienen BCT en vez de BCP
- Creación de un hilo:
  - Crear pila de usuario con argumento y llamada de fin de hilo
  - Crear pila del sistema con dirección inicial del hilo
  - Crear contexto inicial en BCT (Estado = LISTO)
  - Incluir BCT en cola de listos
- Crear proceso: crea hilo implícito

Contexto inicial del hilo



Hilos de usuario

- Hilos creados por biblioteca: SO no conoce su existencia
  - Modelo N:1
- Desventajas:
  - Llamada bloqueante de hilo bloquea todo el proceso
  - No sacan provecho de múltiples procesadores
- Ventaja: gestión de hilos no implica al SO
  - Más ligeros
  - BCT no consume memoria del SO
  - Posibilidad de crear programas con nº muy elevado de hilos
  - Planificación de hilos realizada por biblioteca de usuario
    - Mayor flexibilidad
- Existen esquemas híbridos (Solaris y Mach)
  - Modelo M:N
    - M hilos de usuario sobre N hilos de sistema ($M \geq N$)

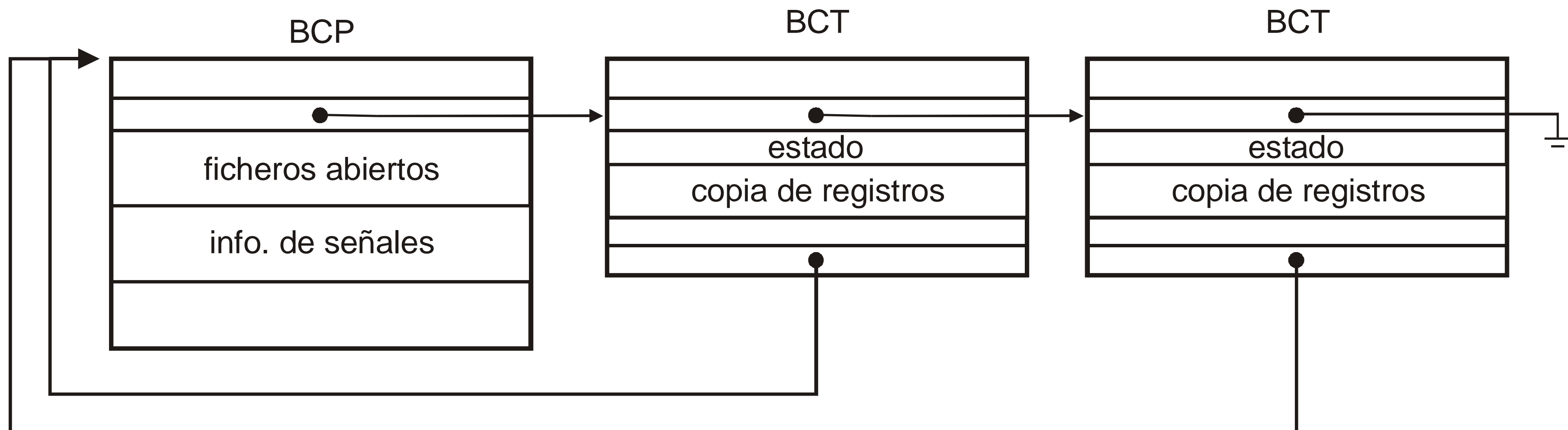
Esquema híbrido de gestión de hilos

- SO gestiona hilos del sistema
- Usuario crea hilos de biblioteca: SO no involucrado
- Biblioteca se encarga de correspondencia entre ambos
  - Biblioteca multiplexa hilos de usuario sobre los de sistema
  - Nº de hilos de usuario (HU) $\geq$ Nº de hilos de sistema (HS)
  - Va creando hilos de sistema según su criterio
    - al menos 1, inicialmente
  - En cada momento: HS ejecuta HU asociado
  - HU hace llamada bloqueante, HS asociado se bloquea
    - Biblioteca puede crear nuevo hilo de sistema
- Biblioteca ajusta dinámicamente nº de HS de manera que:
  - Sean suficientes para ejecutar hilos de usuario activos
  - No gasten muchos recursos del SO
- Planificación realizada por sistema operativo
  - Planificación por biblioteca: *Scheduler Activations*

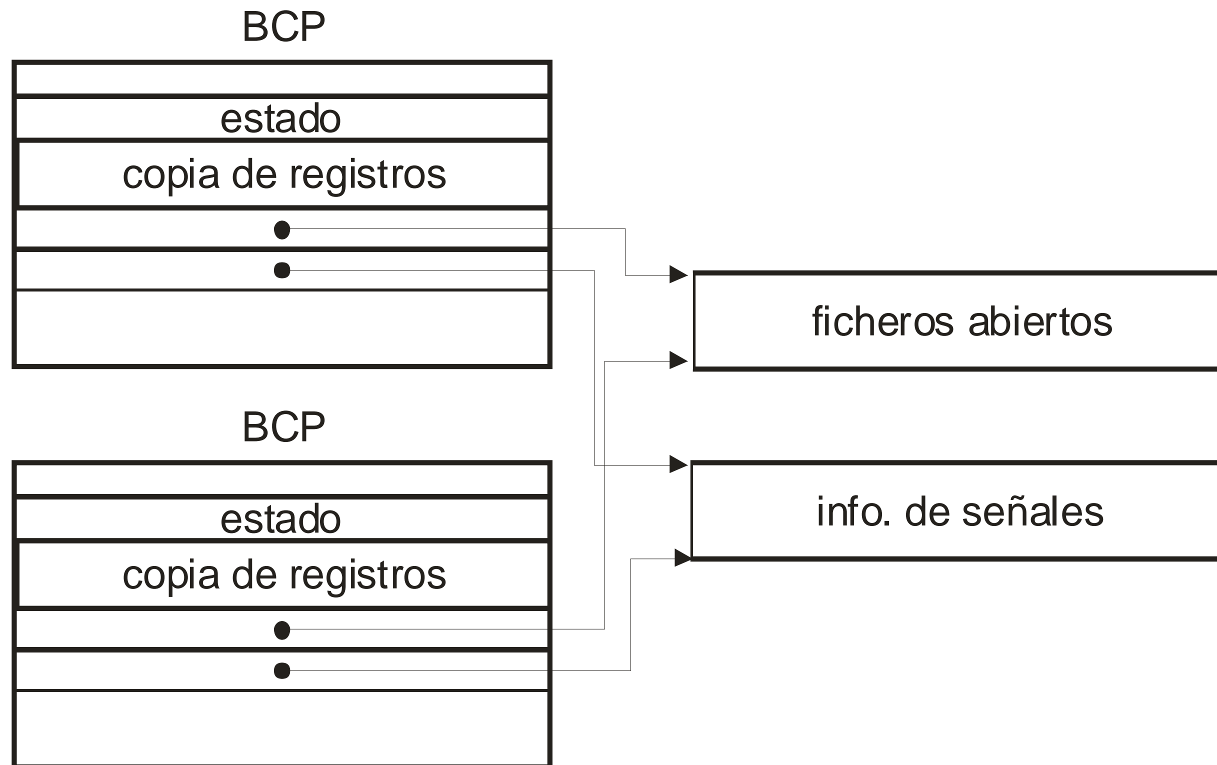
Procesos de peso variable

- En Linux no hay soporte directo de hilos
- CLONE permite especificar qué recursos comparten padre/hijo
  - Info. sobre ficheros (directorio actual, ficheros abiertos, ...)
  - Info. de mapa de memoria
  - Info. de manejo de señales
- *Peso variable*: Más compartimiento → más ligeros
  - Pesado: `Fork` → no compartimiento; recursos duplicados
  - Ligero: Hilo → compartimiento total
- SO gestiona concepto de grupo de hilos
  - Conjunto de procesos que comparten el mismo PID
  - Se tratan de forma integral (excepción en uno, aborta a todos)
- BCP incluye punteros a recursos en vez de los propios recursos
  - No hay BCTs, un BCP por proceso

Implementación nativa de hilos



Implementación basada en procesos ligeros



Clone de *fork* y de *pthread_create*

- *fork*:
  clone(child_stack=0,
  flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, ...)
- *pthread_create*:
  clone(child_stack=0xb7e0c4b4,
  flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, ...)
- Obtenido mediante *strace*

Ejercicio propuesto

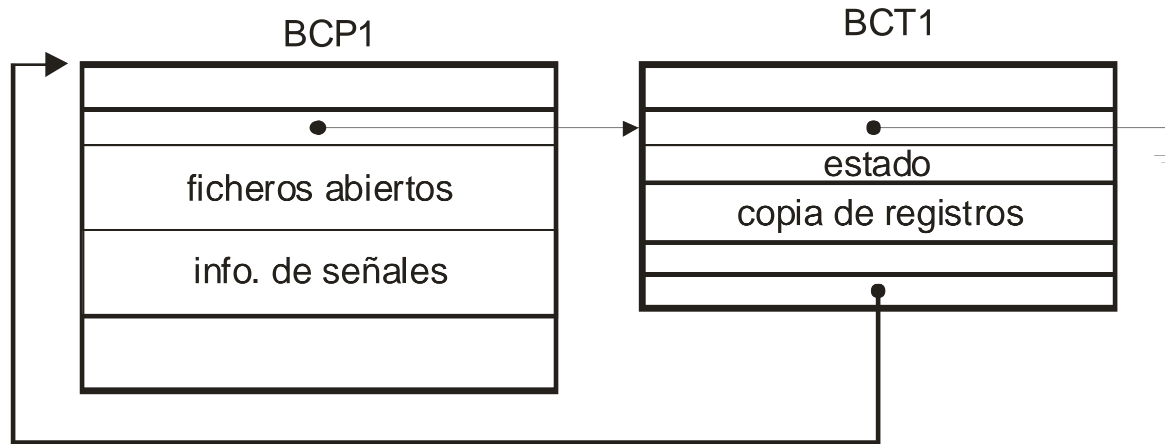
- Dada la siguiente secuencia de ejecución de un proceso:

```
..............
pthread_create(...);
fork();
...............
```

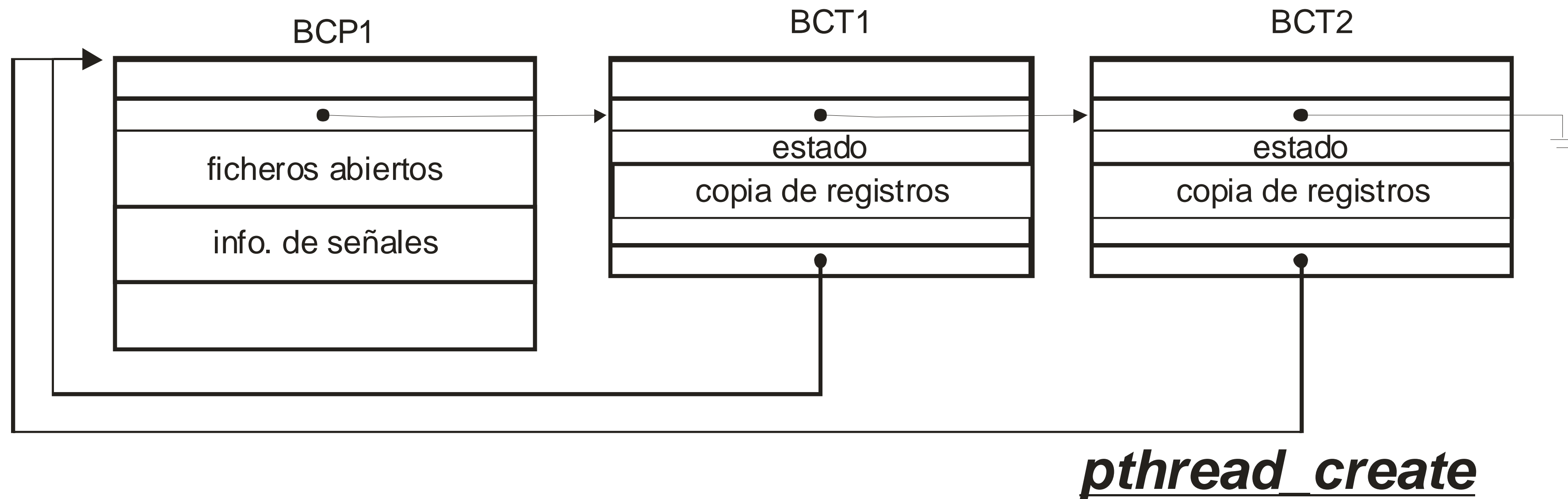
- Dibuje estructuras de datos resultantes para:

1. Implementación nativa
2. Implementación basada en procesos ligeros

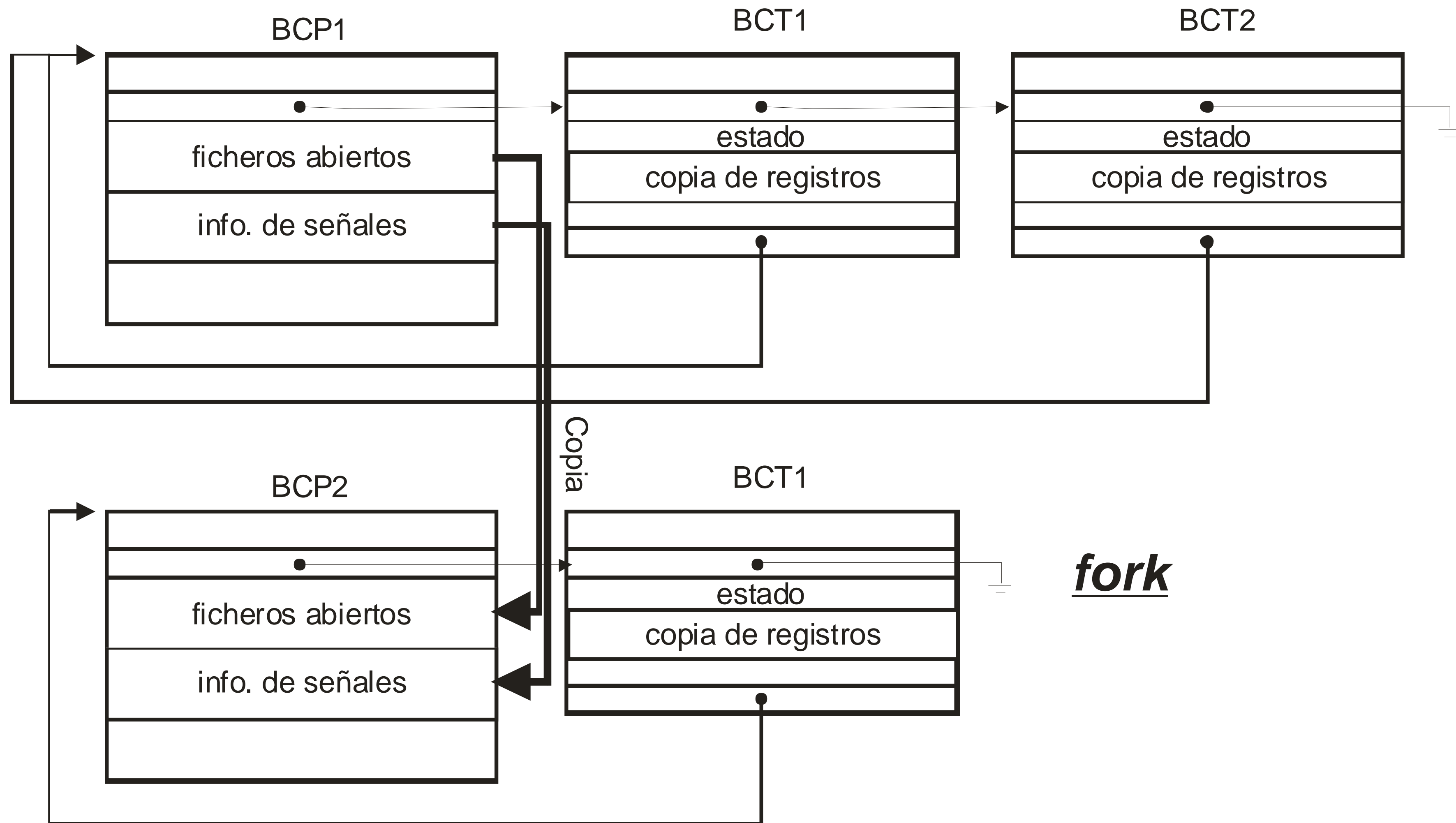
Solución con hilos (1/3)



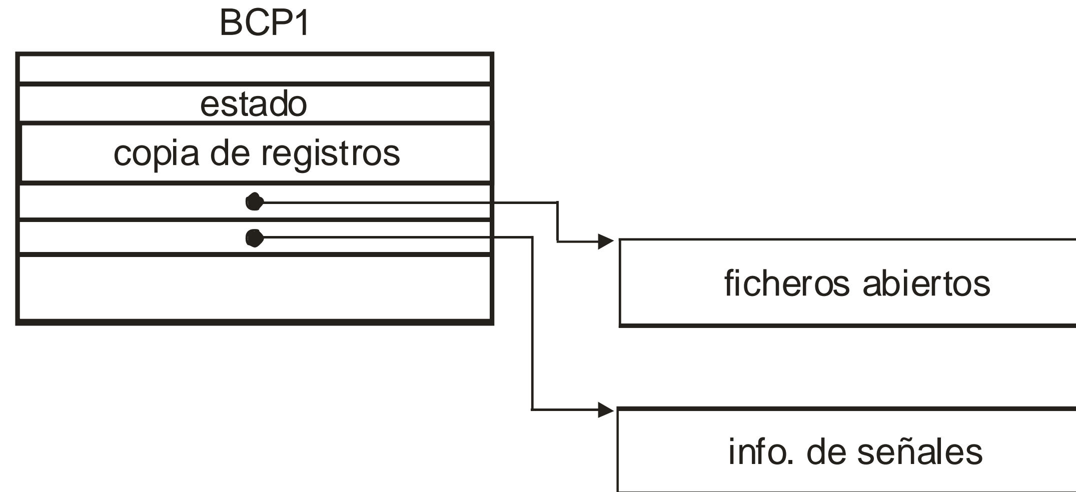
Solución con hilos (2/3)



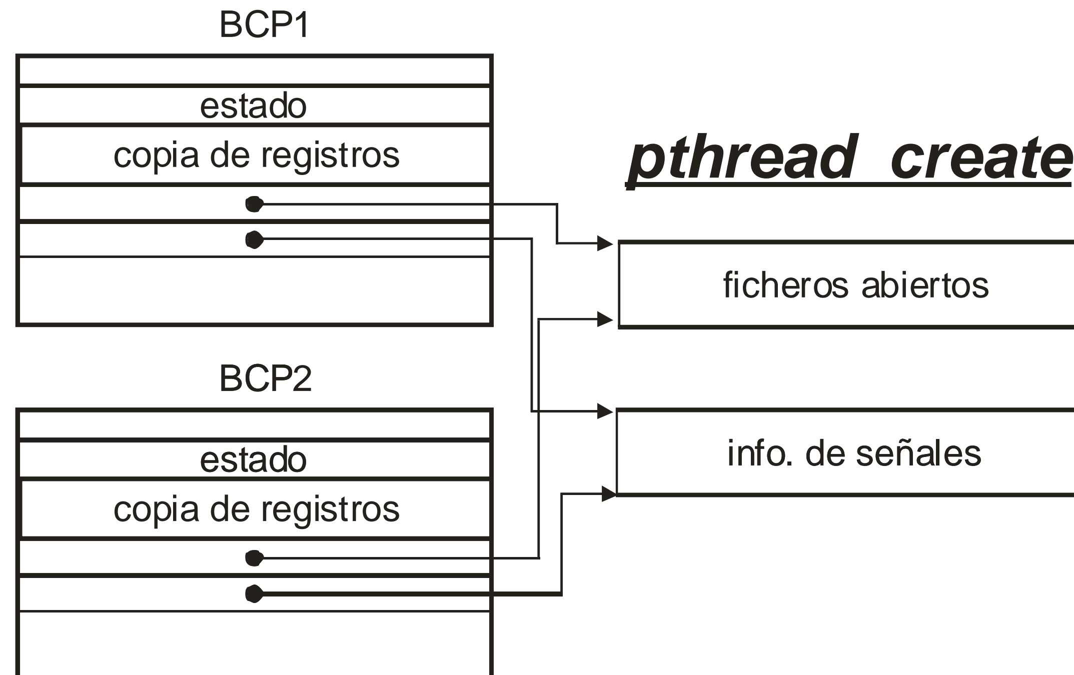
Solución con hilos (3/3)



Solución con procesos ligeros (1/3)



Solución con procesos ligeros (2/3)



Solución con procesos ligeros (3/3)