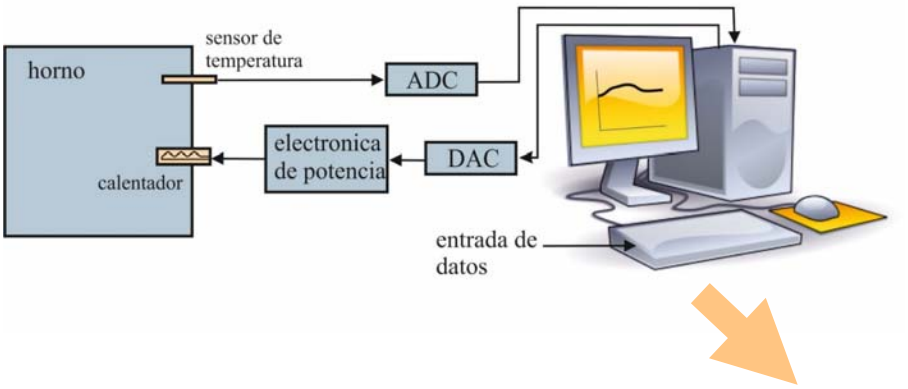
Concurrencia

- Introducción
- Opciones para la concurrencia
- Ejecutivo cíclico
- Programación concurrente
- Procesos en POSIX
- Hilos ("Threads“) POSIX

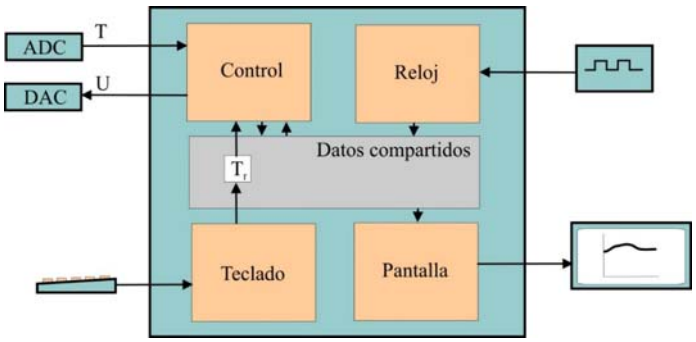
Esquema general

- Introducción
- Opciones para realizar la concurrencia
  - Programa único
  - S.O. monotarea con dos niveles de prioridad
  - Programación concurrente
- Ejecutivo cíclico
- Programación concurrente
  - Concepto, ventajas e inconvenientes
  - Diagrama de estados
  - Relaciones entre procesos
  - Modelos de concurrencia
- Procesos en POSIX
  - Detección de errores
  - Concurrencia en POSIX: procesos e hilos
  - Creación de nuevos procesos
  - Terminación y espera de procesos
- Hilos POSIX (“threads”)
  - Concepto de hilo
  - Creación de un hilo
  - Terminación y espera de hilos
  - Compatibilidad hacia atrás y detección de errores

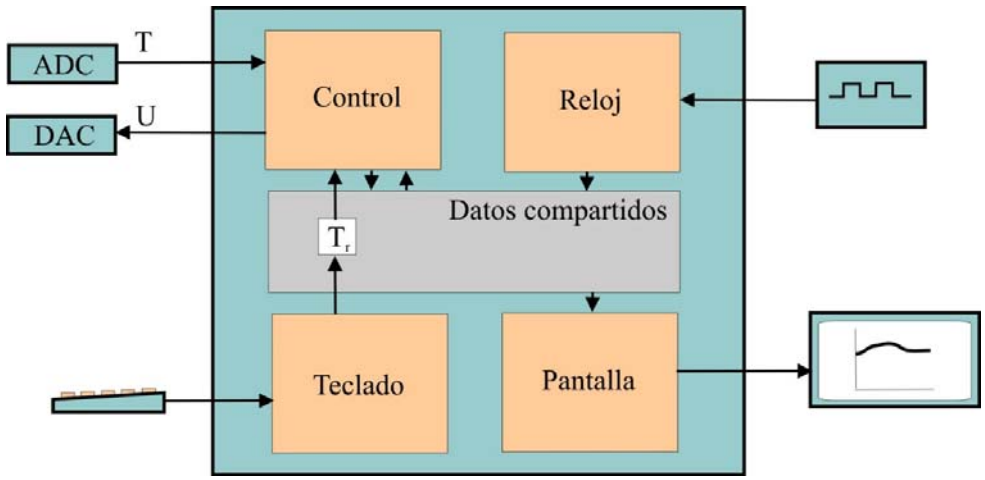
Ejemplo de S.T.R



- Cuatro actividades concurrentes:
  - Reloj de Tiempo Real
  - Bucle de control
  - Interfaz de usuario
  - Representación gráfica



Ejemplo de S.T.R.



Soluciones para concurrencia en S.T.R.

- Más tareas que procesadores: Hay que multiplexar en el tiempo el acceso
- Diferentes soluciones para simular concurrencia:
  - Programa único (ejecutivo cíclico)
    - El programador reparte el tiempo de CPU entre las tareas
    - Las tareas son funciones invocadas por un programa único
  - Solución con dos niveles de prioridad:
    - Alta prioridad (“foreground”): Tareas críticas
    - Baja prioridad (“background”): Tareas no críticas
    - Hay conmutación a “foreground” cuando es necesaria
    - El programador reparte el tiempo de CPU en cada nivel
  - Sistema multitarea (programación concurrente):
    - Tareas definidas separadamente en programas independientes
    - Múltiples niveles de prioridad. El tiempo se asigna indirectamente mediante prioridad y algoritmos de planificación

Soluciones para concurrencia en S.T.R.

- Solución de programa único (ejecutivo cíclico)
  - El programador reparte el tiempo de manera explícita
  - Los bloques son funciones de un programa
  - Comunicación directa a través de datos compartidos

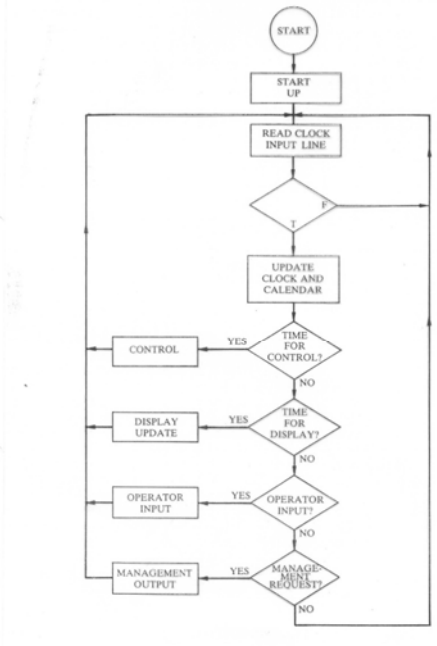


Fig. 5.5 Single program approach.

Soluciones para concurrencia en S.T.R.

- Solución con dos niveles de prioridad
  - Tareas críticas:
    - Reloj de T.R.
    - Control
  - Tareas no críticas:
    - Interfaz de usuario
    - Representación gráfica

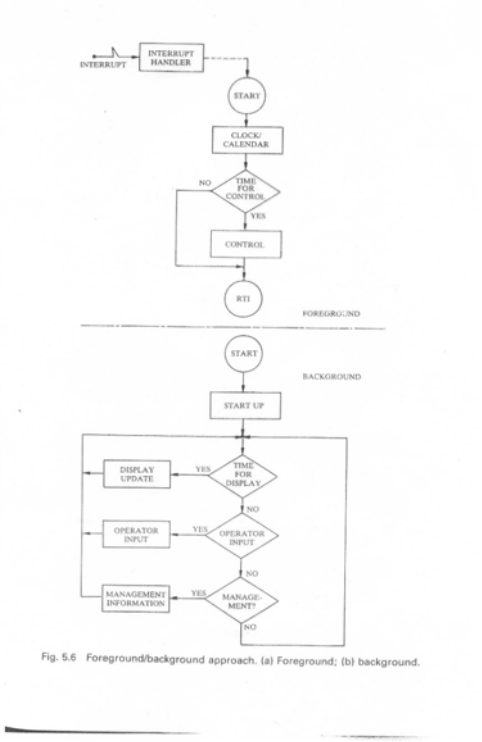


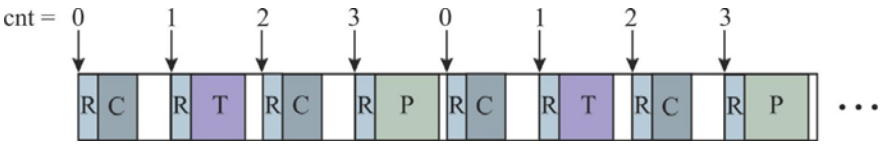
Fig. 5.6 Foreground/background approach. (a) Foreground; (b) background.

Ejecutivo cíclico

- Una implementación:
  - Ciclo secundario: 20 ms
  - Ciclo principal: 80 ms (4 ciclos sec.)
  - Ejecutar Reloj cada ciclo sec.
  - Ejecutar Control en ciclos pares
  - Ejecutar Teclado en ciclo 1
  - Ejecutar Pantalla en ciclo 3

Tarea	T	C
Reloj (R)	20	2
Control (C)	40	8
Teclado (T)	80	12
Pantalla (P)	80	14

Ciclo	Tareas	Coste
0	R,C	10
1	R,T	14
2	R,C	10
3	R,P	16



Ejecutivo cíclico

Pseudocódigo:

```
cnt = 0
Hacer siempre
  Esperar ciclo secundario (20 ms)
  Actualizar cuenta de tiempo (R)
  Si cnt es par, (cada 40 ms), Ejecutar C
    Leer entradas
    Recalcular salidas (PID)
    Actualizar registro de datos
    Actualizar salidas
  Si no,
    Si cnt es 1, (cada 80 ms)
      Si hay entrada del teclado de cambio de parámetros,
        Leerla y actualizar parámetros
        Fin si
      Fin si
      Si cnt es 3, (cada 80 ms)
        Actualizar gráficos de pantalla
      Fin si
    Fin si
  cnt = cnt + 1; Si cnt = 4, cnt = 0 /* Vuelta a empezar */
Fin hacer
```

Más general, basado en tabla:

Hacer siempre
  Hacer para cada entrada **cnt** de la tabla,
    Esperar ciclo secundario (20 ms)
    Ejecutar tareas de la entrada **cnt**
  Fin hacer
Fin hacer

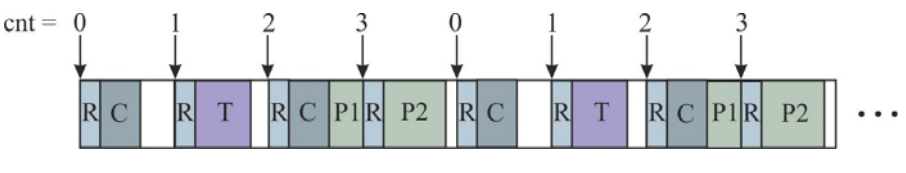
Ciclo	Tareas	Coste
0	R,C	10
1	R,T	14
2	R,C	10
3	R,P	16

Ejecutivo cíclico

- Problemas:
  - Diseño potencialmente complejo
  - Necesidad de rediseño ante modificaciones
  - Poco adecuado para actividades esporádicas
  - A veces es necesario fragmentar actividades
- ¿Qué hacer si Pantalla tarda más?
  - Segmentarlo en P1 (10) y P2 (15) ☹

Tarea	T	C
Reloj (R)	20	2
Control (C)	40	8
Teclado (T)	80	12
Pantalla (P)	80	25

Ciclo	Tareas	Coste
0	R,C	10
1	R,T	14
2	R,C,P1	20
3	R,P2	17

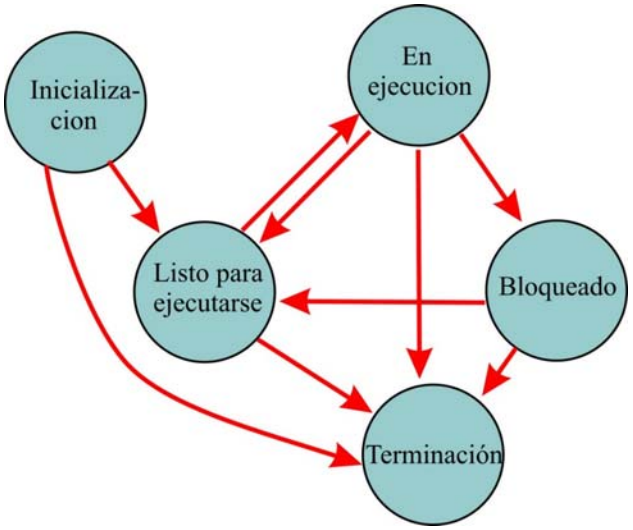


Programación concurrente

- Tareas descritas como programas independientes
- El tiempo se asigna mediante algoritmos genéricos basados en prioridad (planificación – "scheduling")
- Dos opciones:
  - A nivel de lenguaje: Lenguaje concurrente (ADA, JAVA…)
  - A nivel de sistema operativo: Lenguaje secuencial + servicios s.o.
- Ventajas frente a otras soluciones:
  - Modularidad
  - Facilidad de modificación
  - Escalabilidad
  - Portabilidad
  - Protección
- Inconvenientes
  - Menos eficiente: Necesita más recursos que en soluciones de propósito específico (memoria, capacidad de computación…)
  - Son necesarios servicios de comunicación, sincronización y planificación

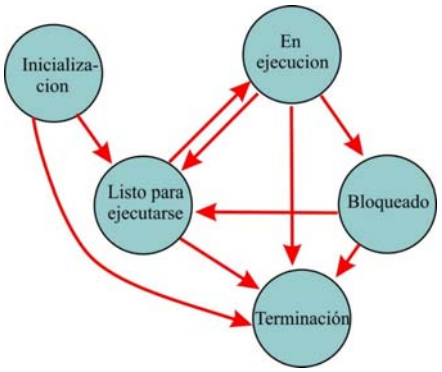
Programación concurrente

- Diagrama de estados para cada tarea
  - Las tareas listas para ejecutarse ("Ready") piden CPU
  - Sólo una (o unas pocas) pueden ejecutarse en un instante dado
  - Las tareas bloqueadas no necesitan CPU; esperan algún evento
- Diagrama independiente del número de procesadores



Programación concurrente

- Principales transiciones de estado:
  - Las tareas en ejecución alcanzan puntos de bloqueo y liberan la CPU
  - Alguna(s) tarea(s) listas para ejecutarse se seleccionan para continuar su ejecución
  - Las tareas bloqueadas se desbloquean y pasan a estar listas para ejecución
  - Las tareas en ejecución pueden pasar a ser
    - "Expulsadas" ("preempted") por tareas más prioritarias
    - Bloqueadas a la espera de un evento
- Planificación: Decisiones sobre el estado de las tareas



Modelos de concurrencia

- Relaciones entre tareas
  - Independencia
  - Cooperación
  - Competencia
- Propiedades de los modelos de concurrencia
  - Estructura: Estática o dinámica
  - Niveles de concurrencia: Plana o anidada
  - Granularidad: Nivel de descomposición en tareas
  - Inicialización: Padre e hijo
  - Terminación: Guardián y dependientes
- Representación de tareas
  - "Fork and join": C+POSIX
  - Declaración explícita: ADA

Modelos de concurrencia

- "Fork and join":
  - Concurrencia "fuera" del lenguaje
  - Creamos un objeto de s.o. y recibimos un identificador ("fork": bifurcación)
  - Podemos esperar su fin ("join": unión)
  - C+POSIX (procesos e hilos)

```
...
c = fork(<tarea nueva>)
…
join(c);
```


Modelos de concurrencia

- Declaración explícita
  - Descripción integrada en el lenguaje
  - Las tareas se declaran como se hace con variables o funciones
  - ADA ("tasks"): La creación y espera de tareas puede estar implícita en el código

```
procedure ejemplo is
    task A;
    task B;
    task body A is
       <declaraciones>
     begin
       <sentencias de A>
     end A;
     <lo mismo para B>
begin
      <sentencias de la
        "procedure" ejemplo>
end ejemplo;
```

Procesos en POSIX

- Detección de errores
- Concurrencia: Procesos e hilos
- Creación de nuevos procesos
- Terminación y espera de procesos

Detección de errores

- La llamada devuelve un código de error
  - Valor “anormal”, normalmente -1
  - En algunos casos, otros valores: NULL, EOF…
- Si hay error, el código de error está en errno
- Cabecera <errno.h>:
  - Declaración de errno
  - Definición de macros de códigos de error
- Para obtener un mensaje: strerror (en <string.h>)
- Algunos errores:
  - EAGAIN: Falta transitoria de recursos
  - EINTR: Función interrumpida por señal
  - EINVAL: Argumento no válido
  - ENOSPC: Medio de almacenamiento sin espacio

Detección de errores

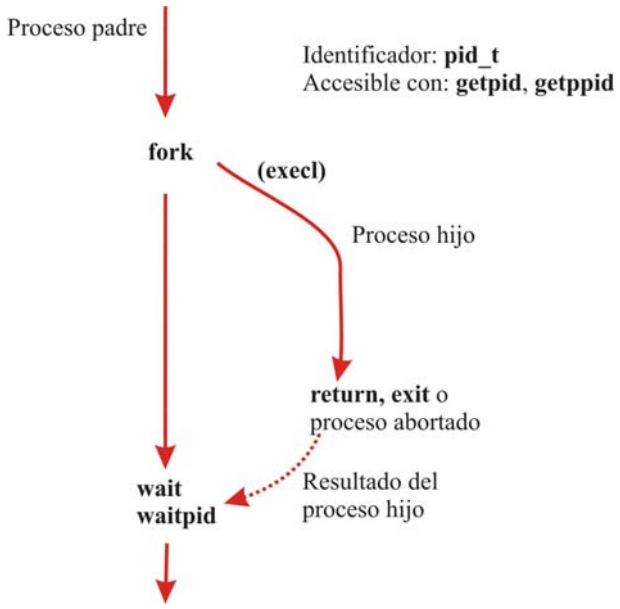
- Un ejemplo:

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
(…)
int res;
res = una_llamada_posix(…);
/* Antes de usar errno, hay que comprobar si ha habido error */
if(res == -1) {
    /* Veamos cual */
    if(errno == EINTR) printf("¡Ha llegado una señal!\n");
    /* Mensaje explicativo, aunque no siempre útil */
    printf("Error: %s\n", strerror(errno));
}
(…)
```


Concurrencia en POSIX

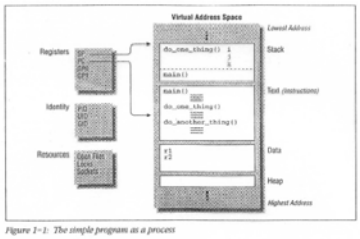
- Dos opciones: Procesos e hilos
- Procesos:
  - Tienen espacios de direccionamiento virtual separados y su propia tabla de páginas
  - Tienen una entrada completa en la tabla de procesos
  - Se asegura la protección
  - No es inmediato compartir memoria
- Hilos:
  - Son siempre parte de un proceso; de algún modo son "subprocesos"
  - Se necesita un mínimo de recursos para crearlos:
    - Comparten la imagen de memoria del proceso
    - Comparten la mayor parte de los recursos del proceso; sólo se añade la mínima información a la tabla de procesos (p.ej. Registros)
  - Ventajas: Eficiencia; comparten memoria
  - Desventajas: No hay protección entre hilos

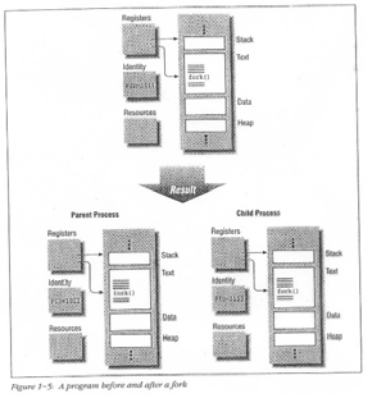
“Fork and join” para procesos POSIX



Creación de un proceso

- Fork:
  - Copia del padre
  - Idéntico estado de ejecución
  - Distinto PID, distinta imagen de memoria
- Fork vuelve dos veces (si todo va bien):
  - Padre: resultado PID del hijo (>0)
  - Hijo: resultado cero
- El padre y el hijo comparten ejecutable. Si no interesa, necesitamos usar **exec**





Creación de un proceso

- **exec**:
  - Familia de funciones: **execl**, **execv**, **execlp**…
  - Permiten superponer un ejecutable sobre la imagen de memoria de un proceso (destruyéndola)
  - Permiten pasar argumentos de línea de comandos
  - Segunda fase (opcional) de la creación de un proceso
- Una de ellas es **execl**:

  int **execl**(const char *ejecutable, const char *arg0, ..., NULL);

  - Si todo va bien, nunca vuelve (o "vuelve" al nuevo **main**)
  - Transforma la imagen de memoria usando el nuevo ejecutable
  - Pasa al nuevo main argumentos de la línea de comandos
  - Sólo habrá un hilo que comienza en **main**
  - Se mantienen algunos atributos (p.ej. tratamiento de señales)

Creación de un proceso

- Sin exec:

```
(…)
    pid_t hijo;
    hijo = fork();
    if(hijo == 0) {  /* Estoy en el hijo */
        funcion_hijo(…);      /* padre e hijo comparten ejecutable */
        exit(0);              /* exit termina el proceso hijo */
    }
    /* Aquí sigue el padre y nunca entra el hijo */
(…)
```

- Con exec:

```
(…)
    pid_t hijo;
    hijo = fork();
    if(hijo == 0) {  /* Estoy en el hijo */
        execl("./hijo, "a", "b", NULL);  /* hijo con ejecutable "./hijo" */
        exit(0);                         /* (por si fracasa execl) */
    }
    /* Aquí sigue el padre y nunca entra el hijo */
(…)
```

Terminación y espera de un proceso

- Terminación:
  - Por iniciativa propia:
    - Con **return** desde **main**
    - Con **exit** o **_exit** desde cualquier función
    - En ambos casos entrega un resultado (o “estado de terminación”) entero de 8 bits
  - Proceso abortado:
    - Causa: Normalmente la llegada de una señal que no se trata
    - No puede entregar realmente un resultado
- Espera de procesos hijos:
  - La espera incluye la recogida del resultado
  - Sólo el padre puede esperar a un hijo, y debe hacerlo (si vive)
  - Dos llamadas:
    - **wait**: Espera a cualquier hijo
    - **waitpid**: Puede esperar a un hijo concreto, o no esperar
  - El entero recibido no es directamente el resultado

Espera de procesos hijos

- Con **wait**:

```
(…)
    pid_t resw, hijo;   /* resultado de wait y pid del hijo */
    int res;            /* resultado del hijo mas info. adicional */
    hijo = fork();
    if(hijo == 0) {
       funcion_hijo(…);
    }
    (el padre sigue ejecutándose)
    resw = wait(&res);    /* Espera (de cualquier hijo, pero sólo hay uno) */
    if(resw != -1) imprime_res(res, resw);  /* Ya veremos qué contiene */
    else  printf(“Error en wait: %s\n”, strerror(errno));
    (…)
```

Espera de procesos hijos

- Con **waitpid**:

```
(…)
pid_t resw, hijo;     /* resultado de waitpid pid del hijo */
int res;              /* resultado del hijo mas info. adicional */
hijo = fork();
if(hijo == 0) {
   funcion_hijo(…);
}
(el padre sigue ejecutándose)
do {
   salir = 1;
   resw = waitpid(hijo, &res, WNOHANG);  /* Mira si el hijo ha acabado */
   if(resw > 0) imprime_res(res, resw);        /* Ya veremos qué contiene */
   else  if(resw == 0)  {
      salir = 0;
      hacer_algo();   /* El padre aprovecha el tiempo… */
   }  else  printf("Error en waitpid: %s\n", strerror(errno));
} while(salir == 0);
```


Espera de procesos hijos

- Función **imprime_res**:

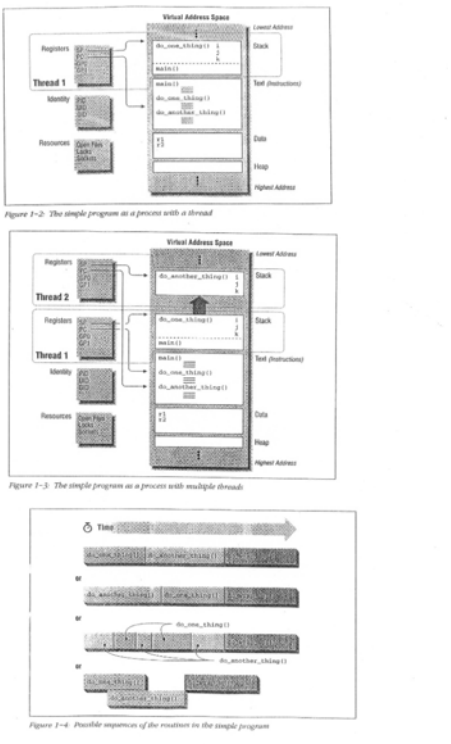
```
void imprime_res(int result, pid_t id) {
     if(WIFEXITED(result)) {
        printf("%ld acabó normalmente con estado %d\n",
                 (long)id, WEXITSTATUS(result));
     }
     else {
        if(WIFSIGNALED(estado)) {
           printf("%ld muerto por señal\n", (long)id);
        }  /* if */
        else {
           /* Otras posibilidades */
        } /* else */
     } /* if */
}
```

Hilos (“threads”)

- Concepto de hilo
- Creación de un hilo
- Terminación y espera de hilos
- Compatibilidad hacia atrás y detección de errores

Concepto de hilo

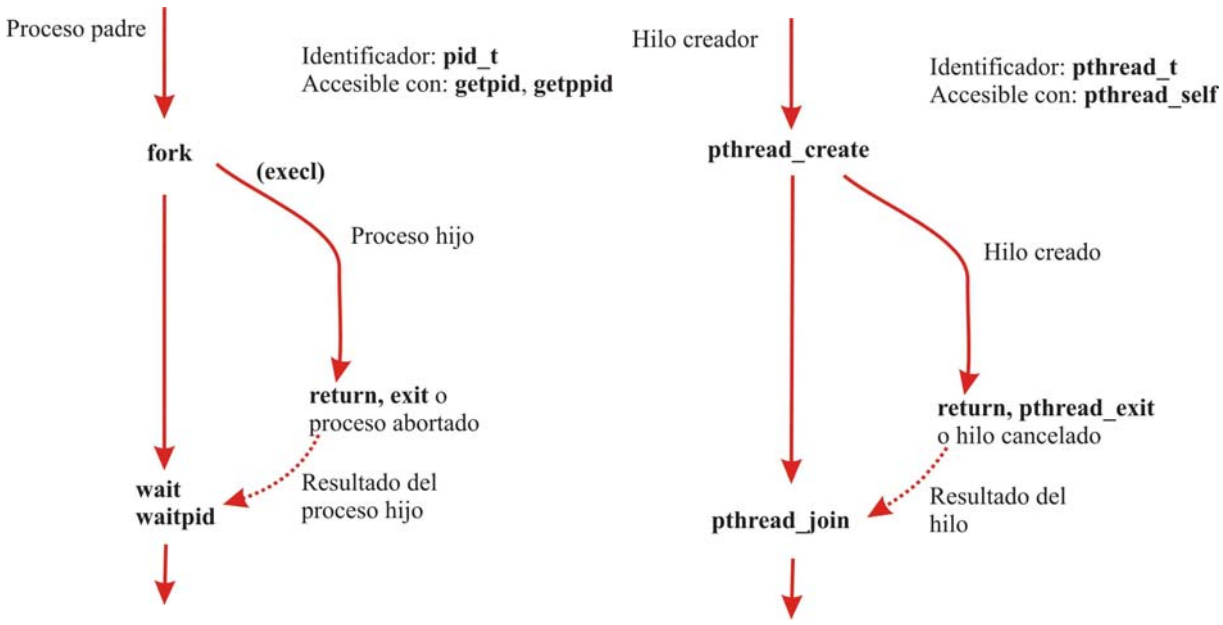
- Todo hilo está asociado a un proceso
- Hilos de un proceso (“threads”):
  - Comparten espacio de direcciones
  - Variables globales accesibles desde todos ellos
  - Recursos compartidos (ficheros, p.e.)
  - No se duplican todos los datos de la tabla de procesos, sólo lo imprescindible
  - No se duplica la imagen de memoria: Sólo se crea una pila nueva
- Ventajas:
  - Creación menos costosa que proceso
  - Conmutación menos costosa (dentro del proceso)
  - Facilidad para compartir memoria
- Desventaja: No existe protección entre hilos



Concepto de hilo

- Concepto existente en la mayor parte de los sistemas operativos (también llamados “procesos ligeros” o “lightweight processes”)
- Los hilos POSIX suelen llamarse “pthreads”
- ¿Cuándo usarlos?
  - Funciones estrechamente relacionadas
  - Dedicar hilos a tratar los diferentes periféricos: así los que necesiten cálculo intensivo no se bloquearán por E/S
  - Eficiencia: Utilizando sólo hilos conseguimos eficiencia a cambio de falta de protección
- Algunos sistemas operativos pueden trabajar sólo con hilos (“thread-only”); por ejemplo, VxWorks – Tornado

“Fork and join” para procesos e hilos



Creación de hilos

- Similar a una llamada a función, pero ésta funciona concurrentemente: Función de arranque
- Llamada **pthread_create**:

  int **pthread_create**(pthread_t ***h**, const pthread_attr_t ***attr**, void *(* **rut_com**)(void *), void ***arg**);

  - Similar a ejecutar **rut_com**(**arg**)
  - Función de arranque: Acepta un puntero genérico y devuelve un puntero genérico
  - Una función de arranque: void ***rhilo**(void *p);
  - El dato apuntado por **arg** no puede ser volátil
  - La estructura **attr** se utiliza para modificar los atributos del hilo (prioridad, dirección y tamaño de la pila…)
  - **pthread_t** no tiene por qué ser un entero
  - No hay relación padre-hijo más allá de la creación del hilo

Espera de procesos hijos

- Creación de un hilo:

```
(...)
void *mirut(void *pg);   /* Prototipo de función de arranque */
(…)
pthread_t  idh;
int *p; p = malloc(2*sizeof(int));                  /* mirut espera una tabla de 2 enteros */
pthread_create(&idh, NULL, mirut, (void *)p); /* Ahora el hilo va en paralelo */
(…)
/* Función de arranque */
void *mirut(void *pg) {
   int *pt = (int *)pg;
   (usa pt[0] y pt[1])
   (…)
   free(pg);                                /* Hay que liberar pg cuando ya no sirva */
   return NULL;                             /* Fin de mirut y del hilo */
}
```

Terminación y espera de hilos

- Terminación:
  - Por iniciativa propia:
    - Con return desde la función de arranque
    - Con **pthread_exit** desde cualquier función
    - En cualquier caso se devuelve un puntero genérico como resultado; el dato al que apunta no debe ser volátil
  - Por cancelación:
    - Otro hilo emite una orden de cancelación con **pthread_cancel**
    - El hilo no puede devolver realmente un resultado
- Espera:
  - Cualquier hilo puede esperar a otro con **pthread_join**:

    int **pthread_join**(pthread_t thread, void **valor);
  - Si el hilo fue cancelado, se recibe PTHREAD_CANCELED, en sustitución del resultado que no ha podido enviar

Espera de procesos hijos

- Creación y espera de un hilo:

```
(...)
void *mirut(void *pg);   /* Prototipo de función de arranque */
(…)
pthread_t  idh; int *result; void *paux;
int *p; p = (int *)malloc(2*sizeof(int));          /* mirut espera una tabla de 2 enteros */
pthread_create(&idh, NULL, mirut, (void *)p); /* Ahora el hilo va en paralelo */
(…)
pthread_join(idh, &paux); result = (int *)paux;
printf("result %d\n", *result); free(paux);
(…)
/* Función de arranque */
void *mirut(void *pg) {
   int *mires;
   (todo como antes)
   mires = (int *)malloc(sizeof(int));  *mires = 18;
   return (void *)mires;                          /* O bien pthread_exit((void *)mires); */
}
```

Cancelación de hilos

- Es similar a abortar un proceso enviándole una señal
- Cualquier hilo puede intentar cancelar a otro con **pthread_cancel**
- La cancelación depende de:
  - Estado de cancelación: Habilitada o deshabilitada
  - Tipo de cancelación:
    - **Asíncrona**: Lo antes posible (puede ser poco recomendable)
    - **Síncrona**: En puntos de cancelación (explícitos o implícitos)
- El hilo puede alterar tanto el estado como el tipo. Por defecto: **Habilitada y síncrona**

Compatibilidad con normas anteriores

- Notificación de errores:
  - Hay una variable indicadora del error por hilo
  - El nombre **errno** hace referencia a la que corresponde al hilo actual
  - Las llamadas **pthread_xx** no tienen por qué usar **errno**. Devuelven el código de error como resultado
- Llamadas seguras en hilos (“pthread-safe”):
  - Sólo las llamadas “pthread-safe” pueden invocarse concurrentemente desde hilos distintos
  - Llamadas no seguras:
    - Habitualmente utilizan alguna variable global
    - Pueden utilizarse si no se invocan de manera concurrente
    - Podrían ser seguras asegurando internamente el acceso exclusivo a datos globales
  - La mayoría de las llamadas 1003.1a y 1003.1b son seguras