Bloque I: Principios de sistemas operativos

Notas:

Tema 2. Concurrencia

- Introducción a la programación concurrente
- Creación e identificación de procesos.
- Ejecución de programas.
- Terminación de procesos y espera a la terminación.
- Threads: conceptos básicos
- Creación de threads y manipulación de sus atributos.
- Terminación de threads.
- Identificación de threads.

1. Introducción a la programación concurrente

Muchos problemas se expresan de forma natural mediante varias actividades concurrentes:

- sistemas de control atendiendo a múltiples subsistemas y eventos
- sistemas multicomputadores o distribuidos
- para el uso concurrente de múltiples recursos

La concurrencia implica prever la sincronización:

- para la utilización de recursos y datos compartidos
- y para el intercambio de eventos e información

Definiciones

Un *flujo de control* o *thread* es una secuencia de instrucciones de programa.

Un *programa concurrente* es aquel con múltiples flujos de control, que generalmente cooperan entre sí.

Los flujos de control se llaman *procesos*, *tareas*, o *threads*, dependiendo del contexto.

A veces, dos o más flujos de control requieren un orden particular para sus operaciones: *sincronización*.

Para ejecutar un programa concurrente se necesita un *planificador* que gestione cuándo se ejecuta cada flujo de control y qué recursos del sistema pueden usar.

Ejemplo: Control de un Automóvil

Actividades a controlar:

Medida de Velocidad
C=4 ms.
T=20 ms.
D=5 ms.

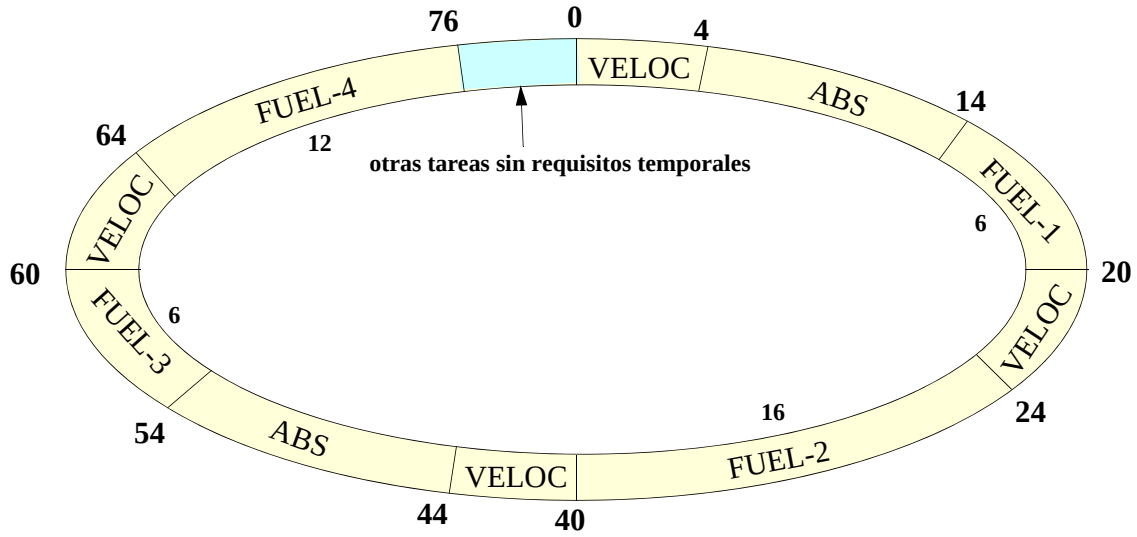
Control de Frenos ABS
C=10 ms.
T=40 ms.
D=40 ms.

Control de Inyección
C=40 ms.
T=80 ms.
D=80 ms.

C = Tiempo de ejecución
T = Periodo
D = Plazo máximo de finalización

Ejemplo: solución cíclica

Una solución cíclica implica partir la actividad más larga en varias partes:



Ejemplo: solución concurrente

La solución concurrente es más fácil de diseñar y modificar:

Medida de la velocidad

```
loop
   Medida_Velocidad;
   next:=next+0.02;
   Sleep_until next;
end loop;
```

Control de frenos ABS

```
loop
   Frenos_ABS;
   next:=next+0.04;
   Sleep_until next;
end loop;
```

Control de inyección

```
loop
   Inyeccion_Fuel;
   next:=next+0.08;
   Sleep_until next;
end loop;
```

Actividades sin requisitos temporales

```
loop
   hacer cálculos;
   ...
end loop;
```


Ejemplo: mantenimiento

Suponer que necesitamos ejecutar una actividad aperiódica que tiene 1ms de tiempo de ejecución. El tiempo mínimo entre llegadas es de 80 ms, pero el plazo es de 20 ms:

1. Solución cíclica:
   - Muestrear al menos cada 19 ms para ver si hay que ejecutar
   - Esto implica partir ABS y FUEL en varios trozos cada uno
2. Solución concurrente:
   - Añadir un nuevo proceso con alta prioridad y repetir el análisis de planificabilidad
   - No se necesitan modificaciones al código existente.

Declaración de procesos: soporte en el lenguaje

Algunos lenguajes tienen instrucciones especiales para crear procesos concurrentes:

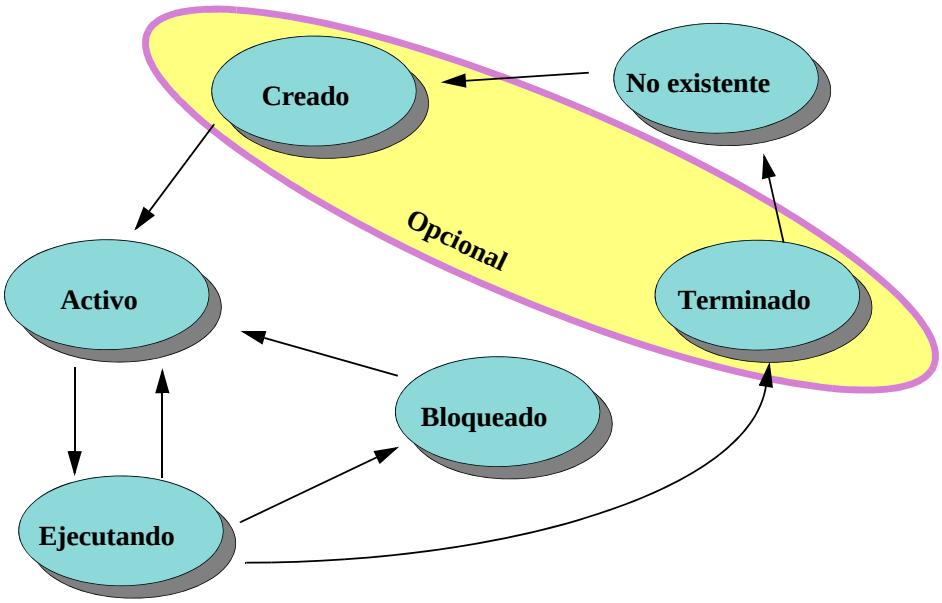
- instrucción *estructurada*: *cobegin, coend* (Pascal Concurrente)
- instrucción de *bifurcación*: *fork*
- *declaración* de un objeto de la clase "proceso" en la parte declarativa del programa (Ada)
- *creación dinámica* de un objeto de la clase "proceso" (Java)

Declaración de procesos: soporte en el sistema operativo

Cuando la concurrencia no está soportada en el lenguaje se pueden usar servicios de un sistema operativo:

- operación de bifurcación: *fork* crea una copia del proceso padre; posteriormente la copia puede ejecutar otro programa (*exec*)
- operación *spawn*: se crea un nuevo proceso concurrente que ejecuta el programa o función indicados

Estados de un proceso



2. Creación e identificación de procesos

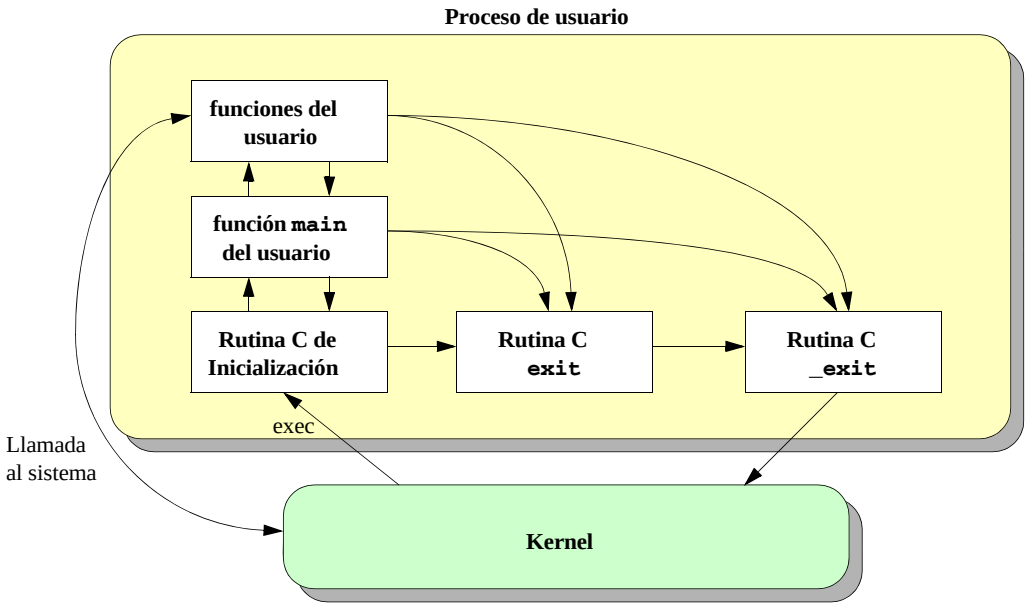
Identificador de proceso (pid):

- es un entero positivo que identifica a un proceso

Parentesco de procesos:

- el padre de un proceso es el proceso que lo creo
- los procesos creados por el padre son hijos de éste

Estructura de un proceso en C



Ejemplo de un programa C

```
// El programa muestra en pantalla todos sus argumentos
#include <stdio.h>

int main (int argc, char *argv[])
{
   int i;

   // Print all the arguments
   for (i=0;i<argc;++i)
   {
      printf("%s\n",argv[i]);
   }
   exit(0);
}
```

La llamada *fork*

La llamada *fork* crea un nuevo proceso (hijo) que es una copia exacta del padre, excepto por lo siguiente:

- **el *pid* del hijo es diferente**
- **el hijo tiene su propia copia de los descriptores de fichero, que apuntan a las mismas descripciones de fichero que en el padre**
- **el hijo no hereda:**
  - **alarmas**
  - **temporizadores**
  - **operaciones de I/O asíncronas**

El hijo hereda el estado del padre (incluyendo el PC), sus semáforos, objetos de memoria, política de planificación, etc.

La llamada *fork* (cont.)

Interfaz:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

Retornos:

- ***fork* devuelve al proceso hijo un cero**
- ***fork* devuelve al proceso padre el *pid* del hijo**
- **en caso de error, devuelve -1, no crea el hijo, y la variable *errno* se actualiza al valor asociado al error**

Llamadas para identificación del proceso:

```
pid_t getpid(void);     pid_t getppid(void);
```

Ejemplo de uso de *fork*

```
// El proceso crea un hijo; padre e hijo muestran pids
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
   pid_t childpid;

   if ((childpid=fork()) == -1) {
        perror("can't fork");
        exit(1);
   } else if (childpid == 0) {
        // child process
        printf("child: own pid = %d, parent pid = %d\n",
             getpid(), getppid());
        //the parent may be dead
        exit(0);
   } else {
```


Ejemplo de uso de *fork* (cont.)

```
        // parent process
        printf("parent: own pid = %d, child pid = %d\n",
                getpid(), childpid);
        exit(0);
   }
}
```

3. Ejecución de un programa: *exec*

La familia de llamadas *exec* sustituyen la imagen de proceso actual por una nueva imagen de proceso, que se halla en un fichero

Cuando la nueva imagen de proceso es un programa C, debe tener la siguiente función:

```
int main (int argc, char *argv[]);
```

Los parámetros corresponden a los argumentos en la llamada

Asimismo, la siguiente variable contiene un puntero a un array de punteros que apuntan a los strings de entorno:

```
extern char **environ;
```

La llamada a *exec* sólo retorna en caso de error

Ejecución de un fichero: *exec* (cont.)

Los descriptores de ficheros abiertos en el proceso original continúan abiertos en el proceso nuevo, excepto aquellos que tengan el flag FD_CLOEXEC habilitado

El nuevo proceso hereda:

- el *pid* y *ppid* del proceso original
- la política de planificación
- alarmas y señales pendientes

Las llamadas *exec*

```
int execl (const char *path, const char *arg,...);
int execv (const char *path, char *const argv[]);
int execle (const char *path, const char *arg,...);
int execve (const char *path, char *const argv[],
      char *const envp[]);
int execlp (const char *file, const char *arg,...);
int execvp (const char *file, char *const argv[]);
```

El argumento ***arg**, y sucesivos, son arg0, arg1, arg2,...,NULL

El último argumento en *execle* será:

```
char *const envp[]
```

Si no existe *envp*, se toma el entorno del proceso original

Ejemplo de uso de *exec* (cont.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
   pid_t childpid;
   char *av[3];

   if ((childpid=fork()) == -1) { // process creates child
        perror("can't fork");
        exit(1);
   } else if (childpid == 0) {
        // child process executes the ps command
        av[0] = "ps";
        av[1] = "-A";
        av[2] = (char *)0;
        execv("/bin/ps",av);
        perror("execv failed");
        exit(1);
```

Ejemplo de uso de *exec* (cont.)

```
    } else {
          // parent process
          printf("parent continues here and stops");
          exit(0);
    }
}
```


4. Terminación de un proceso: *exit*

Dos tipos de terminación:

- normal: retorno de *main*(), *exit*(), o *_exit*()
- anormal: *abort*(), o señal que "mata" el proceso

La llamada *_exit* devuelve al sistema los recursos usados, excepto el *pid* y el estatus de salida si el padre existe

Interfaz:

```
void _exit (int status);
```

La llamada *exit* nunca retorna

Cuando un proceso termina, el *ppid* de sus hijos se hace igual al de un proceso del sistema (generalmente 1 = *init*)

Espera a la terminación de un proceso: *wait*

Las llamadas *wait* permiten esperar a la terminación de un proceso hijo y obtener su estatus de salida

Interfaz:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *stat_loc);
pid_t waitpid (pid_t pid, int *stat_loc,
                    int options);
```


Las llamadas *wait*

La llamada *wait*() suspende al padre hasta que se puede obtener el estatus de salida de uno de sus hijos

- la función retorna el *pid* del hijo
- en el entero al que apunta *stat_loc* se devuelve 0, si el estatus de salida es 0, o información que permite conocer:
  - el estatus de salida
  - si el hijo terminó a causa de una señal, y qué señal
  - etc.

La llamada *waitpid*() permite especificar el *pid* del hijo al que se quiere esperar, y diversas opciones:

- no suspenderse
- etc.

Ejemplo de llamada *wait*

```
// parent waits for child termination; child spawns a new process
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
   pid_t childpid;
   pid_t grandchildpid;
   char *av[3];
   int status;

   if ((childpid=fork()) == -1) {
         perror("can't fork");
         exit(1);
   } else if (childpid == 0) {
         // child process
         printf ("spawning...\n");
```


Ejemplo de llamada *wait* (cont.)

```
         if ((grandchildpid=fork()) == -1) {
               perror ("child can't fork");
               exit (1);
         } else if (grandchildpid != 0) {
               // child process continues here and exits
                printf("child will exit now\n");
                exit (0);
         } else {
               // grandchild process execs here
               // notice that its parent pid is 1
               sleep(3);
               av[0] = "top";
               av[1] = "b";
               av[2] = (char *)0;
               execv("/usr/bin/top",av);
               perror("execv failed");
               exit(1);
         }
   } else {
```

Ejemplo de llamada *wait* (cont.)

```
        // parent process
        childpid=wait(&status);
        printf("child finished with status %d\n",status);
        printf("now parent may do other things\n");
        exit(0);
    }
}
```


5. Threads: conceptos básicos

Thread:

- un flujo de control simple perteneciente a un proceso
- tiene un identificador de thread (*tid*)
- el *tid* sólo es válido para threads del mismo proceso
- tiene su propia política de planificación, y los recursos del sistema necesarios, tales como su propio stack, etc
- todos los threads de un proceso comparten un único espacio de direccionamiento

Proceso en una implementación multi-thread:

- un espacio de direccionamiento con uno o varios threads
- inicialmente contiene un solo thread: el thread principal

Threads: conceptos básicos (cont.)

Creación de un proceso (*fork*):

- se crea con un único thread
- si el proceso que llama es multi-thread sólo puede hacer llamadas "seguras", hasta realizar un *exec*

Ejecución de un programa:

- todos los threads del proceso original se destruyen
- se crea también con un único thread

Servicios bloqueantes:

- en una implementación multithread, sólo se suspende el thread que invoca el servicio

Threads: conceptos básicos (cont.)

Los threads tienen dos estados posibles para controlar la devolución de recursos al sistema:

- "*detached*" o independiente: `PTHREAD_CREATE_DETACHED`
  - cuando el thread termina, devuelve al sistema los recursos utilizados (tid, stack, etc)
  - no se puede esperar la terminación de este thread con `pthread_join`()
- "*joinable*" o sincronizado: `PTHREAD_CREATE_JOINABLE`
  - cuando el thread termina, mantiene sus recursos
  - los recursos se liberan cuando el thread termina, y otro thread llama a `pthread_join`()
  - este es el valor por defecto

6. Creación de threads

Para crear un thread es preciso definir sus atributos en un objeto especial

El objeto de atributos

- debe crearse antes de usarlo: *pthread_attr_init*()
- puede borrarse: *pthread_attr_destroy*()
- se pueden modificar o consultar atributos concretos del objeto (pero no los del thread, que se fijan al crearlo)

Los atributos definidos son:

- tamaño de stack mínimo (opcional)
- dirección del stack (opcional)
- control de devolución de recursos ("detach state")

Atributos de creación: interfaz

```
#include <pthread.h>
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);

int pthread_attr_setstacksize (pthread_attr_t *attr,
                               size_t stacksize);
int pthread_attr_getstacksize (const pthread_attr_t *attr,
                               size_t *stacksize);

int pthread_attr_setstackaddr (pthread_attr_t *attr,
                               void *stackaddr);
int pthread_attr_getstackaddr (const pthread_attr_t *attr,
                               void **stackaddr);

int pthread_attr_setdetachstate (pthread_attr_t *attr,
                                 int detachstate);
int pthread_attr_getdetachstate (const pthread_attr_t *attr,
                                 int *detachstate);
```

Llamada para creación de threads

Interfaz:

```
int pthread_create (pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *),
        void *arg);
```

Esta función:

- crea un nuevo thread con atributos especificados por *attr*.
- devuelve el *tid* del nuevo thread en *thread*
- el thread se crea ejecutando *start_routine(arg)*
- si *start_routine* termina, es equivalente a llamar a *pthread_exit* (observar que esto difiere del thread *main*)

Ejemplo de creación de threads

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

// Thread que pone periódicamente un mensaje en pantalla
// el periodo se le pasa como parámetro
void *periodic (void *arg) {
   int period;

   period = *((int *)arg);
   while (1) {
      printf("En el thread con periodo %d\n",period);
      sleep (period);
   }
}
```

Ejemplo de creación de threads (cont.)

```
// Programa principal que crea dos threads periódicos
int main ()
{
   pthread_t th1,th2;
   pthread_attr_t attr;
   int period1=2
   int period2=3;
   // Crea el objeto de atributos
   if (pthread_attr_init(&attr) != 0) {
        printf("error de creación de atributos\n");
        exit(1);
   }
   // Crea los threads
   if (pthread_create(&th1,&attr,periodic,&period1) != 0) {
        printf("error de creación del thread uno\n");
        exit(1);
   }
```


Ejemplo de creación de threads (cont.)

```
   if (pthread_create(&th2,&attr,periodic,&period2) != 0) {
        printf("error de creación del thread dos\n");
        exit(1);
   }
   // Les deja ejecutar un rato y luego termina
   sleep(30);
   printf("thread main terminando\n");
   exit (0);
}
```

7. Terminación de threads

Función para terminación de threads:

```
#include <pthread.h>
void pthread_exit (void *value_ptr);
```

Esta función termina el thread y hace que el valor apuntado por *value_ptr* esté disponible para una operación *join*

- se ejecutan las rutinas de cancelación pendientes
- al terminar un thread es un error acceder a sus variables locales
- cuando todos los threads de un proceso se terminan, el proceso se termina (como si se hubiera llamado a *exit*)

Terminación de threads (cont.)

Se puede esperar (*join*) a la terminación de un thread cuyo estado es sincronizado (*joinable*), liberándose sus recursos:

```
#include <pthread.h>
int pthread_join (pthread_t thread,
                  void **value_ptr);
```

También se puede cambiar el estado del thread a “detached”, con lo que el thread, al terminar, libera sus recursos:

```
#include <pthread.h>
int pthread_detach (pthread_t thread);
```

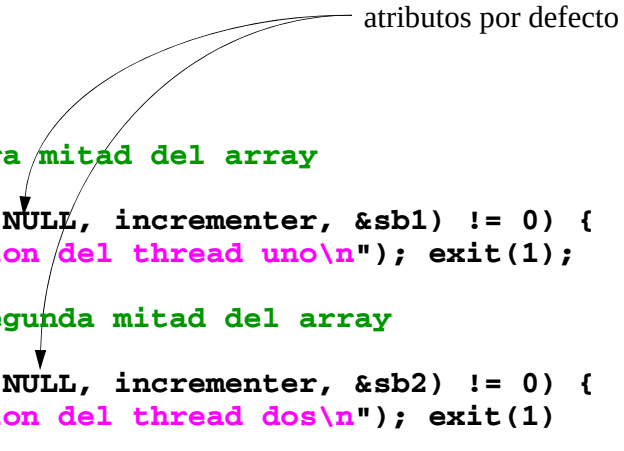
Existen también funciones para cancelar threads, habilitar o inhibir la cancelación, etc. (ver el manual).

Ejemplo

```
#include <pthread.h>
#include <stdio.h>
#define MAX   500000
#define MITAD 250000

typedef struct {
  int *ar;
  long n;
} subarray;

// Thread que incrementa n componentes de un array
void * incrementer (void *arg) {
  long i;

  for (i=0; i< ((subarray *)arg)->n; i++) {
    ((subarray *)arg)->ar[i]++;
  }
  pthread_exit(NULL);
}
```


Ejemplo (cont.)

```
// programa principal que reparte el trabajo de incrementar
// los componentes de un array entre dos threads
int main() {
  int ar [MAX];
  pthread_t th1,th2;
  subarray sb1,sb2;
  void *st1, *st2;
  long suma=0, i;

  sb1.ar = &ar[0];// primera mitad del array
  sb1.n = MITAD;
  if (pthread_create(&th1, NULL, incrementer, &sb1) != 0) {
    printf("error de creacion del thread uno\n"); exit(1);
  }
  sb2.ar = &ar[MITAD];// segunda mitad del array
  sb2.n = MITAD;
  if (pthread_create(&th2, NULL, incrementer, &sb2) != 0) {
    printf("error de creacion del thread dos\n"); exit(1)
  }
```



Ejemplo (cont.)

```
  // sincronizacion de espera a la finalizacion

  if (pthread_join(th1, &st1) != 0) {
    printf ("join error\n"); exit(1);
  }
  if (pthread_join(th2, (void **)&st2) != 0) {
    printf ("join error\n"); exit(1);
  }
  printf ("main termina; status 1=%d; status 2=%d\n",
          (int) st1, (int) st2);

  for (i=0; i<MAX; i++) {
     suma=suma+ar[i];
  }
  printf ("Suma=%d\n",suma);
  exit(0);
}
```


8. Identificación de threads

Identificación del propio thread:

```
pthread_t pthread_self(void);
```

Comparación de *tids*:

```
int pthread_equal (pthread_t t1, pthread_t t2);
```