Threads

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Noviembre 2009

Procesos

El programa es lo que resulta de la compilación de un código en algún lenguaje, es código objeto estática.

El proceso es lo que resulta realmente de la ejecución de un programa, es decir, no sólo el código objeto sino todo lo que necesita el programa para correr ("contexto").

En particular:

- puede haber varios procesos ejecutando el mismo código objeto de un programa dado;
- aún después de haber lanzado un mismo programa, se puede tener varios procesos corriendo en el sistema.

Procesos

- La noción de contexto del proceso está intrinsecamente dependiente y ligada al sistema de explotación (OS).
- El OS maneja, para un proceso dado:
  - la asignación de memoria en el sistema para un nueva instancia de un programa (el proceso),
  - el acceso a recursos físicos en la maquina,
  - el acceso compartido al procesador entre todos los procesos.

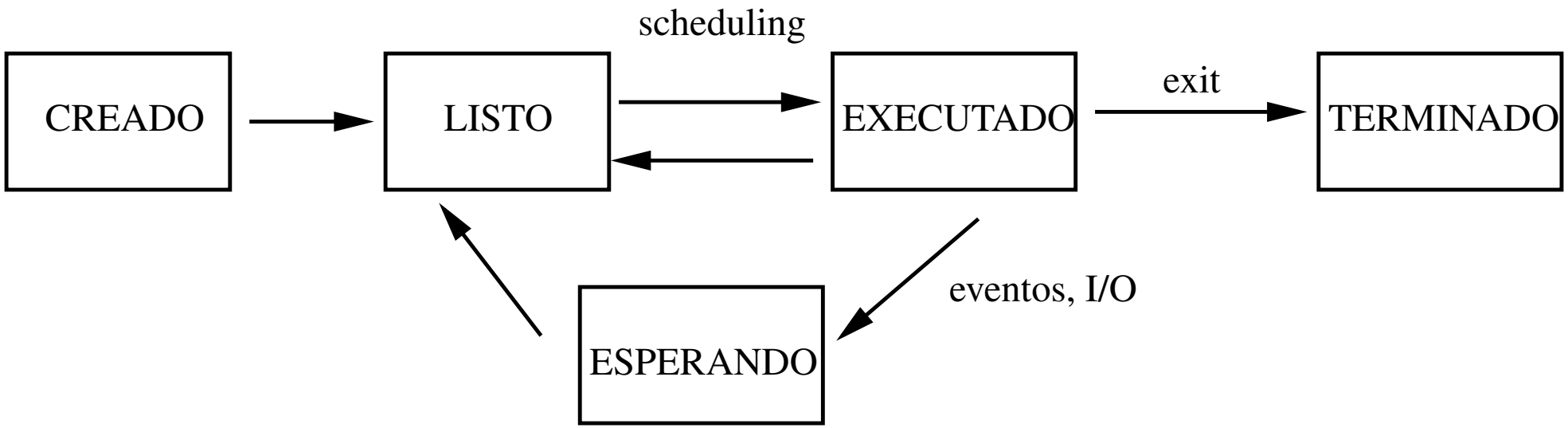
Procesos

- En la mayoría de los OS, el sistema es multitasking, es decir que da la "ilusión" al usuario de que varios procesos se están ejecutando simultáneamente.
- En práctica, muchas veces hay nada más un procesador, lo *multitasking* no corresponde a una realidad hardware, sino software: muy frecuentemente, el sistema asigna un poco de tiempo de calculo a un proceso.
- Sistema *multitasking* prioritario : el procesador automáticamente da la mano al OS para atribuir tiempo de calculo a otros procesos, en función de prioridades asignadas a cada proceso.

"Tiempo compartido".

Procesos

- El *scheduling*, es decir, lo de repartir el tiempo entre los diferentes procesos, es una de las operaciones más fundamentales de un OS.
- Típicamente eso lleva a definir para un proceso varios estados.



Procesos

No hay estándar en cuanto a la definición de esos estados, pero en general existen estados para:

- el estado siguiendo el cargo del código objeto de un soporte físico (disco duro...) a la memoria viva ("creado"),
- el estado correspondiendo al esperar que el *scheduler* elija al proceso para estar ejecutado en el procesor ("listo"),
- el estado correspondiendo a la ejecución de las instrucciones de código por el procesador ("ejecutado"),
- el estado especial en que el proceso tiene que esperar a unos recursos como input del usuario ("esperando"),
- el estado siguiendo la terminación del programa, en que nada más falta descargar el código objeto de la memoria ("terminado").

Procesos

- Dados todos los procesos correspondiendo a aplicaciones que el usuario quiere correr el trabajo del OS es de asignar esos procesos por intervalos de tiempo chiquitos a el o los diferentes procesadores: scheduling.
- Típicamente, colas múltiples de procesos en esperanza de ser "servidos" para tiempo de CPU, con prioridades.
- Problema de optimización,
  - utilizar lo máximo posible la CPU,
  - maximizar el número de procesos terminados por unidad de tiempo,
  - para cada proceso, minimizar el tiempo de ejecución,
  - para cada proceso, minimizar el tiempo de presencia en la cola de espera,
  - para cada proceso, minimizar el tiempo de respuesta (i.e. el proceso no se acabó pero empieza a dar resultados)...

Procesos: “contexto”

En consecuencia el “contexto” de un proceso puede incluir un número importante de datos

1. Manejo de la memoria
   - Puntos de acceso : direcciones de la pila, montículo, . . .
2. Manejo de los archivos
   - I/O : archivos abiertos. . .
   - Directorio corriente. . .
   - Identidad del usuario, grupo. . .
3. Manejo del proceso
   - Estado : listo, ejecutado. . .
   - Registros : PC(program counter), PSW(program state word)
   - Identidad : PID, identidad del proceso padre. . .
   - Scheduling : prioridad. . .

Procesos

- Aunque hoy las máquinas usan generalmente varios procesadores, este mecanismo de tiempo compartido está usado para distribuir de manera equilibrada los procesos entre los procesadores.
- Sistemas SMP (symmetric multiprocessing): varios procesadores idénticos conectados a través de memoria compartida.
- Multi-processing "flojo". Máquinas conectadas a través de red muy rápida ("cluster").
- Multi-processing "apretado" (CMP: chip-level multi-processing). Limitación física al número de procesadores, pero mucho más eficiente (memoria propia al CPU + memoria compartida cache).
  Ex: Core2 Duo, Xeon, AMD X2...

Procesos: conmutación de contexto

- Operación básica que hace pasar de un proceso a otro (p a q).
- Implica guardar/restaurar el estado de un proceso en el sistema, típicamente al menos
  - Guardar el contexto de p en algún lugar en memoria (por ejemplo en la pila de p).
  - Encontrar el contexto de q en memoria (en la pila de q).
  - A partir del contexto, restaurar el proceso q: registros, instrucción que ejecutar...
- Puede ser costosa (depende del número de registros implicados...).
- Existe conmutación "hardware" (interna al CPU) pero pocos OS usan la funcionalidad.

Procesos

Una manera de implementar paralelismo es asociar a un programa un primer proceso y dejar que este proceso crea otro proceso (proceso "hijo"), haciendo una parte del problema.

Una función de UNIX para eso: FORK().

Procesos: fork()

- FORK() crea un nuevo proceso cloneado (mismo código) a partir del proceso que llama esa función.
- Los dos procesos (el padre y el hijo) continúan su ejecución deespués de la llamada.
- En caso de error, la función regresa un valor negativo y no se crea el proceso hijo.
- Si OK, el proceso padre (el que hizo el fork()) recupera el pid (process ID) del proceso hijo (> 0) y el proceso hijo recupera un pid 0.

Procesos: fork()

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    int pid = 0, status = 0;
    if ((pid = fork()) < 0) {
      exit(1);
    }
    if (pid == 0) { /* Son */
      printf(``My father's pid %d\n'', getppid());
      exit(1);
    } else { /* Father */
      printf(``My PID, %d, and my son's %d\n'', getpid(), pid);
      wait(&status);
      printf(``\n Child process ended %d\n'', status);
      exit(0);
    }
}
```

Procesos: fork()

- Proceso hijo creado, copia conforma de su papa, excepto por su PID y por la direcciones en memoria...
- Los dos procesos bifurcan al nivel del FORK().
- En función del valor de PID (> 0 para el padre), podemos ejecutar tal o tal parte del código (pero no compartimos nada !).
- El WAIT() hace que el padre espera la terminación del proceso hijo (sin eso creamos un ZOMBIE).

Procesos

Limitaciones para la paralelización

- Conmutación de contexto cuello de botella.
- Sincronización de los procesos muy difícil para cálculos en paralelo.
- Manejar memoria compartida es posible (IPC en UNIX) pero bastante complicada que usar en el código; si no pasar por I/O.

Procesos "pesados".

Procesos ligeros

Los procesos ligeros son una creación destinada a facilitar la paralelización

- comparten memoria común, accesible muy simplemente en el código,
- dan herramientas de sincronización.

Procesos ligeros

En breve, para utilizar *threads*:

- Haces tu programa tradicionalmente, con variables globales, una función MAIN, ...
- Haces sub-programas adentro de funciones.
- El proceso que corresponde al programa estará compuesto por varios *threads*: el que corresponde al main y todos los que el programa principal puede crear.
- Los *threads* se ejecutan en paralelo, y comparten recursos (en particular la memoria correspondiendo a las variables globles).

Procesos ligeros

Qué es un *thread*:

- una versión "ligera" de un proceso que ejecuta alguna función,
- tiene una cosa específica: su pila, para los datos locales, llamadas a sub-funciones,...
- comparte la memoria estática y la memoria montículo con los *threads* nacidos del mismo proceso,
- es mucho más eficiente la conmutación de contexto entre dos *threads* del mismo proceso que entre dos procesos normales ("pesados").

Procesos ligeros

Aplicación típica en que se puede requerir paralelización: un arreglo de datos que se puede procesar en dos partes independientes.

```
int datos[10000];
void main() {
 doPart1();
 doPart2();
}
```

Procesos/threads:

```
 /* Con procesos */
 pid=fork();
 if (pid==0) {doPart1();}
 else {
  pid=fork();
  if (pid==0) {doPart2();}
 }
```

```
/* Con threads */
pthread_create (doPart1,..
pthread_create (doPart2,..
```

Procesos ligeros

Lo benéfico:

- Más fácil de usar que los procesos normales.
- Comparten memoria.
- Conmutación de contexto rápida.
- Varias herramientas de sincronización.
- Relativamente simple que diseñar las aplicaciones corriendo en paralelo.

Lo más problemático:

- No hay estándar unificado!

Procesos ligeros POSIX

Un *thread* POSIX es definido al nivel del OS por:

- un identificador único y una prioridad,
- un acceso a recursos sistema,
- una pila,
- un resguardo de los registros,
- dots

Procesos ligeros POSIX

PTHREAD_CREATE	Función que crea un thread.
PTHREAD_EXIT	Sale del thread (en cambio de EXIT que sale del proceso y de todos los threads).
PTHREAD_JOIN	Suspende la ejecución del thread que la llama hasta que el thread cuyo id es pasado en argumento se termine.
PTHREAD_SELF	Regresa el id del thread que la llama.

Procesos ligeros POSIX

```
#include <pthread.h>
 int pthread_create(pthread_t *thread,
   const pthread_attr_t *attr,
   void *(*start_routine)(void *), void *arg);
```

Ejemplo:

```
void *doPart1 (void *);
...
pthread_t idSon1;
...
pthread_create (&idSon1, NULL, doPart1, NULL);
```

Procesos ligeros POSIX

El segundo argumento contiene los atributos (propiedades) del thread. Con NULL se usan los atributos por default, sino:

```
#include <pthread.h>
pthread_attr_t tattr;
pthread_attr_init(&tattr);
...
pthread_create (&idSon, &tattr, doPart1, NULL);
```

Permite en particular definir la prioridad asociada, el tamaño de la pila...

Procesos ligeros POSIX

- Los threads son creados por default con la misma prioridad que la del proceso creador.
- Esa prioridad se puede cambiar.
- Con prioridades iguales al proceso creador, eso deja éste hacer unas instrucciones hasta que los otros sí empiecen.
- Para controlar la activación de un thread, contar no sobre su prioridad sino mejor sobre mecanismos de sincronización.

```
int main (void) {
    ...
    pthread_create (&idSon1, ...);
    pthread_create (&idSon2, ...);
    ...
}
```

Procesos ligeros POSIX

- La paralelización la define entonces el usuario a través de la definición adecuada de funciones encargadas de "una parte del trabajo".
- Para el OS, el trabajo de *scheduling* es el mismo: ve el proceso ligero como un proceso normal.
- La conmutación de contexto está más sencilla.
- Hay trabajo fuerte para el programador de manejar, para las diferentes funciones que asocia a sus threads, el acceso concurrente a recursos, en particular a la memoria común.

Procesos ligeros POSIX: peligros

```
static int v = 0;
void *doSomething(void *) {
 int i;
 pthread_t t = pthread_self();
 for (i=0; i<50000; i++) v++;
 printf("%d: v = %d\n ", t, v);
}
int main (void) {
 pthread_t thread1, thread2;
 pthread_create (&thread1 ,NULL,doSomething, NULL);
 pthread_create (&thread2, NULL,doSomething, NULL);
 pthread_join (thread1, NULL);
 pthread_join (thread2 ,NULL);
 printf ("v = %d\n ", v);
 return 0;
}
```

Procesos ligeros POSIX: peligros

- El valor de *v* puede ser diferente de 100000.
- Por ejemplo:
  - en thread1 se pone v en un registro,
  - en thread1 se incrementa el registro,
  - en thread2 se pone v en un registro,
  - en thread2 se incrementa el registro,
  - en thread1 se copia el registro a v,
  - en thread2 se copia el registro a v.
- Resultado: una sola incrementación.

Procesos ligeros POSIX

Para los threads POSIX, existen varias formas de sincronización:

- El candado ("lock" o "mutex") para acceso exclusivo (exclusión mutual).
- Variables condicionales.
- Semáforos.

Procesos ligeros POSIX: candados

PTHREAD_MUTEX_INIT	Crea un candado (inicialmente no cerrado).
PTHREAD_MUTEX_DESTROY	Destruye el candado.
PTHREAD_MUTEX_LOCK	Toma el candado y lo cierra, si está libre, sino bloquea el thread.
PTHREAD_MUTEX_TRYLOCK	Igual que el precedente, pero no bloquea el *thread* si el candado no está libre.
PTHREAD_MUTEX_UNLOCK	Regreso del candado por el *thread* que lo ha tomado.

Procesos ligeros POSIX

Ejemplo precedente:

```
static int v = 0;
pthread_mutex_t lock;
void *doSomething(void *) {
 int i;
 pthread_t t = pthread_self();
 for (i=0; i<50000; i++) {
     pthread_mutex_lock(&lock);
     v++;
     pthread_mutex_unlock(&lock);
 }
 printf("%d: v = %d\n ", t, v);
}
```

Procesos ligeros POSIX

Ejemplo precedente:

```
int main (void) {
  pthread_t thread1, thread2;
  pthread_mutex_init(&lock,NULL);
  pthread_create (&thread1,NULL,doSomething, NULL);
  pthread_create (&thread2, NULL,doSomething, NULL);
  pthread_join (thread1, NULL);
  pthread_join (thread2,NULL);
  printf ("v = %d\n", v);
  return 0;
}
```

Procesos ligeros POSIX

- Para que sí funcione con la ilusión de paralelización, es importante que el tiempo durante el cual un *mutex* está tomado no sea demasiado grande !
- Tiene que ser el mismo *thread* que toma (LOCK) el *mutex* que el que lo libera (UNLOCK)

Procesos ligeros POSIX

PTHREAD_COND_INIT	Crea una variable condicional.
PTHREAD_COND_DESTROY	Destruye una variable condicional.
PTHREAD_COND_SIGNAL	Libera un *thread* bloqueado sobre una variable condicional.
PTHREAD_COND_BROADCAST	Libera todos los *threads* bloqueados sobre una variable condicional.
PTHREAD_COND_WAIT	Bloquea el thread y libera el mutex (luego lo recupera cuando se desbloquea).
PTHREAD_COND_TIMED_WAIT	Igual que el precedente pero bloquea solo por un periodo de tiempo.

Procesos ligeros POSIX

Queremos hacer algo cuando se cambia un valor de variable *v*

```
int v;
void set(int newv) {    // Executed in thread 1
    pthread_mutex_lock(& lock);
    v = newv;
    pthread_mutex_unlock(&lock);
}
void waitEquality(int vv){ // Executed in thread 2
    while (1) {
        pthread_mutex_lock(&lock);
        int localv = v;
        pthread_mutex_unlock(&lock);
        if (localv == vv) break;
        // Here, something may happen
        // including 2 calls to set!!
    }
```

Procesos ligeros POSIX

Problema: con esa solución a base de *mutex* podemos perder una actualización de las que tenemos como meta (la de v) ya que pueden occurir dos set en el espacio de tiempo no "cerrado" por los mutex (en la segunda, por ejemplo, el test no pasaría).

Solución: variable condicional !

Procesos ligeros POSIX

```
int v;
pthread_cond_t condv;

void set(int newv) {    // Executed in thread 1
     pthread_mutex_lock(& lock);
     v = newv;
     pthread_cond_broadcast(&condv);
     pthread_mutex_unlock(&lock);
}
void waitEquality(int vv){ // Executed in thread 2
        pthread_mutex_lock(&lock);
        while (v!=vv)
          pthread_cond_wait(&condv, &lock);
        pthread_mutex_unlock(&lock);
}
```

Procesos ligeros POSIX

El efecto de PTHREAD_COND_WAIT:

- libera el *mutex* pasado como segundo argumento (entonces se puede escribir en *v*),
- bloquea el thread que lo llama (aquí el segundo) hasta que otro *thread* envíe un SIGNAL o BROADCAST.
- El SIGNAL desbloquea nada más un *thread*, el de más alta prioridad bloqueado sobre la variable condicional, el BROADCAST desbloquea todos los threads bloqueados sobre esta variable condicional.

Procesos ligeros POSIX

Semáforos.

SEM_INIT	Crea un semáforo.
SEM_DESTROY	Destruye un semáforo.
SEM_WAIT	Espera que el semáforo tenga un valor positivo, y lo decrementa.
SEM_TRYWAIT	Igual sin la espera.
SEM_POST	Incrementa el valor del semáforo.
SEM_GETVALUE	Recupera el valor del semáforo.

Procesos ligeros POSIX

Ejemplo: una aplicación en que datos se almacenan y se usan a través de dos *threads*. El *thread* principal:

```
sem_t emptyPlaces;
sem_t fullPlaces;
int begin, end;
int datos[size];
sem_init(& emptyPlaces,size);
sem_init(& fullPlaces,0);
begin = 0;
end   = size;
```

Procesos ligeros POSIX

Un primer *thread* que genera datos y les almacena en la memoria común:

```
void set(int dato) {
  // Wait for at least one free space
  sem_wait(&emptyPlaces);
  end =(end +1)%size;
  datos[end]=dato;
  sem_post(&fullPlaces);
}
```

Procesos ligeros POSIX

Un segundo *thread* que procesa esos datos:

```
int process() {
 int d;
 // Wait for at least one data
 sem_wait(&fullPlaces);
 d=datos[begin];
 begin =(begin +1)%size;
 sem_post(&emptyPlaces);
 return d;
}
```

Procesos ligeros POSIX

- El semáforo es un contador que bloquea con el WAIT() hasta que recupere un valor positivo (por un POST()) en algún otro lado.
- De una manera, el *mutex* es una forma binaria de semáforo.
- Concepto de sincronización presente en la mayoría de los OS en particular los orientados a tiempo real.

Procesos ligeros: otras API

```
/**
 * Thread class for updating connectors
 */
class updateConnectorsThread : public QThread {
 private:
  vddrStructure& vStruct;
 public:
  updateConnectorsThread(vddrStructure& vStruct);
  virtual void run();
};
```

Procesos ligeros: otras API

```
/**
 * Constructor
 */
updateConnectorsThread::updateConnectorsThread(vddrSt
vStruct(vStruct){
}
```

Procesos ligeros: otras API

```
/**
 * Overload of run
 */
void updateConnectorsThread::run() {
  // Update the structure
  vStruct.updateCommonVis(-1);
  vStruct.updateConnectors(-1);
}
```

Procesos ligeros: otras API

QThreads: interfaz multi-plataformas, dan funcionalidades equivalentes a los pthreads (mutex, variables condicionales, semáforos...)

```
updateC->wait();
updateC->start();
```

Procesos ligeros: Java

- Todo a través de clases/objetos.
- Principio similar a los QThread: heredar de la clase Thread y sobrecargar el método run().
- Usar el método start() para empezar un Thread.
- Un interfaz corresponde a esos objetos: Runnable; es posible de hecho no heredar de Thread sino implementar Runnable.

Procesos ligeros: Java

```
class PrettyThread extends Thread {
    public void run(){
     System.out.println("Running␣"+ getName());
    }
}
public static void main (String args[]) {
   Thread idSon1 = new PrettyThread("Son1");
   Thread idSon2 = new PrettyThread("Son2");
   idSon1.start();    // Method run() of idSon1
   idSon2.start();     // Method run() of idSon2
}
```

Procesos ligeros: Java

```
class PrettyRunnable implements Runnable {
    String tname;
    public PrettyRunnable (String s) {tname = s;}
    public void run() {
     System.out.println("Running " + tname);
    }
}
void main (String args[]) {
    PrettyRunnable idSon1 = new PrettyRunnable("Son1")
    PrettyRunnable idSon2 = new PrettyRunnable("Son2")
    new Thread(idSon1).start();
    new Thread(idSon2).start();
}
```

Procesos ligeros: Java

Métodos de Thread

START()	activa el *thread*,
RUN()	ejecutado por start(),
SLEEP(MS)	bloquea el *thread* durante un tiempo en ms,
SETPRIO(P)	da una prioridad al *thread*,
YIELD()	da la mano (a un *thread* de prioridad igual o justo inferior a ese),
JOIN()	espera el fin del *thread*,
JOIN(MS)	espera el fin del *thread* durante un tiempo en ms.

Procesos ligeros: Java

El mismo problema de al rato:

```
class Dato {
 int v;
 public Dato(int vv) {v=vv;};
 public void increment(int i) {v=v+i;}
}
class PrettyThread extends Thread {
 static Dato d = new Dato (0);
 public void run () {
      for (int i = 0; i<50000; i++)
          d.add(1);
 }
}
```

Procesos ligeros: Java

El mismo problema de al rato:

```
static void main (String args[]) {
    Thread idSon1 = new PrettyThread ("Son1");
    Thread idSon2 = new PrettyThread ("Son2");
    idSon1.start ();
    idSon2.start ();}
```

La operación add no es “elemental” del punto de vista del “procesador”, entonces pueden ocurrir varias combinaciones de atribuciones de recursos en que los add no son separados.

Procesos ligeros: Java

Aquí todos los objetos tienen un mutex implícito. Está cerrado el candado al

- llamar un método synchronized
- entrar en un bloque synchronized

```
class Dato {
 int v;
 public Dato(int vv) {v=vv;};
 public synchronized void increment (int i) {
    v=v+i;
 }
 public void sub (int i) {
    synchronized (this) {
     v=v-i;
    }
 }
```