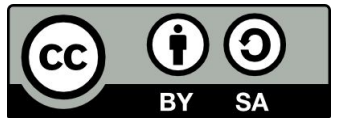
Computación de Altas Prestaciones

Modelo de programación basado en memoria compartida

José Luis Risco Martín
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid



Índice

Bibliografía:

- Grama, A. Gupta, G. Karypis, V. Kumar. An Introduction to Parallel Computing, Design and Analysis of Algorithms. Addison-Wesley, 2003.
- Jesús Carretero, Sistemas Operativos – Una visión aplicada. McGraw-Hill. 2007.

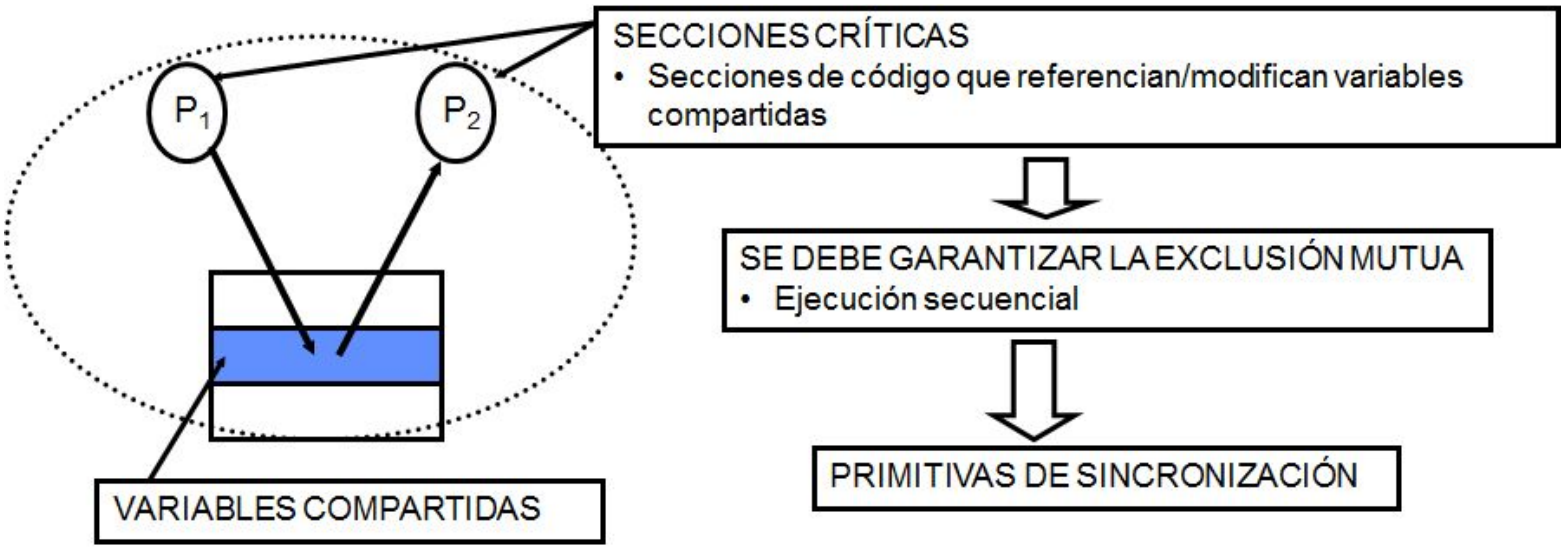
Introducción (1/2)

- Un sistema multiprocesador donde todos los procesos pueden ejecutarse con el mismo privilegio se denomina multiprocesador simétrico (SMP)
- Todos los recursos se comparten (memoria y dispositivos entrada/salida)
- Un sistema operativo soporta multiprocesamiento simétrico siempre que se puedan proteger regiones críticas (garantía de exclusión mutua)
- El planificador distribuye los procesadores disponibles entre los procesos (threads)preparados para ejecutarse
- El paralelismo aparece cuando un programa ha sido codificado (o compilado) de modo que fragmentos del mismo se ejecutan como procesos (threads) independientes.

Thread = proceso ligero (solo contexto hardware)

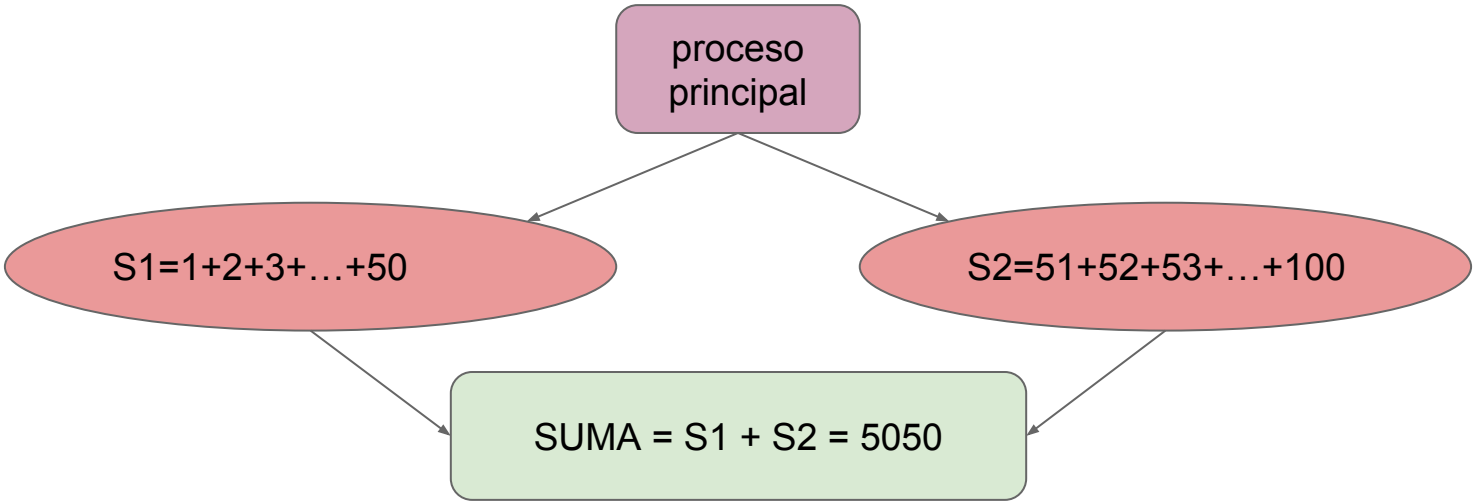
Introducción (2/2)

- La memoria es compartida y puede ser accedida por cualquier proceso/thread envuelto en la computación
- Computación basada en sincronización para garantizar la consistencia de los datos
- La comunicación se realiza mediante compartición de los datos



Problema de Consistencia de una Variable Compartida

- Supongamos un sistema compuesto por n hebras
- Cada uno tiene un fragmento de código que accede/modifica un recurso compartido:
  - Sección crítica
- Queremos que sólo una de las hebras en cada instante pueda ejecutar su sección crítica
- **Ejemplo:**



Problema de Consistencia de una Variable Compartida

- **Ejemplo (cont.):**
  - Supongamos un sistema compuesto por n hebras Calculamos la suma de los N primeros números usando hebras

```
int suma_total = 0; // Var compartida
void suma_parcial(int ni, int nf) {
  int j = 0;
  int suma_parcial = 0; // Var. privada
  for (j = ni; j <= nf; j++)
    suma_parcial = suma_parcial + j;
  suma_total = suma_total + suma_parcial;
  pthread_exit(0);
}
```

  - Si varios hilos ejecutan concurrentemente este código se puede obtener un resultado incorrecto.

Problema de Consistencia de una Variable Compartida

- **Ejemplo (cont.):**
  - Posible implementación en ensamblador del cálculo de suma_total:

```
suma_total = suma_total + suma_parcial;
```

```
LDR R1, suma_total   #R1=0 (la 1ª vez)
LDR R2, suma_parcial #R2=1275
ADD R1,R1,R2         #R1=1275
STR R1, suma_total   #suma_total=1275
```

Problema de Consistencia de una Variable Compartida

- **Ejemplo (cont.):**
  - Posible situación de conflicto:

```
LDR R1, suma_total   #R1=0
LDR R2, suma_parcial #R2=1275
############## Cambio de contexto ##############
LDR R1, suma_total   #R1=0
LDR R2, suma_parcial #R2=3775
ADD R1, R1, R2       #R1=3775
STR R1, suma_total   #suma_total=3775
############## Cambio de contexto ##############
ADD R1, R1, R2       #R1=1275
STR R1, suma_total   #suma_total=1275
```

Problema de Consistencia de una Variable Compartida

- **Ejemplo (cont.):**
  - Posible solución del conflicto:
    - Solicitar permiso para entrar en la sección crítica
    - Indicar la salida de sección crítica

```
void suma_parcial(int ni, int nf) {
  int j = 0;
  int suma_parcial = 0;
  for (j = ni; j <= nf; j++)
    suma_parcial = suma_parcial + j;
  /** <Entrada en la sección crítica> **/
    suma_total = suma_total + suma_parcial;
  /** <Salida de la sección crítica> **/
  pthread_exit(0);
}
```

Funciones de Sincronización de Alto Nivel (1/3)

- Garantizan el acceso secuencial a las secciones críticas
- Proporcionadas por el SO e implementadas en todos los lenguajes de alto nivel
- **Ejemplos (rutinas atómicas e indivisibles)**
  - Cierres (locks)
  - Semáforos
  - Barreras

CIERRES

Variable cierre: *flag* => ON(1) o OFF(0)

Funciones:

Lock(flag):

Si *flag* == ON (activado) entonces espera a que *flag* == OFF

Si *flag* == OFF (desactivado) entonces *flag* = ON

Unlock(*flag*):

flag = OFF

```
P_padre: SUM = 0
         for k =1 to N do
                  Fork P_hijo(k)
         end for
```

```
P_hijo:  Lock (flag)
         SUMA = SUMA + A(k)
         Unlock(flag)
         Join
```

Funciones de Sincronización de Alto Nivel (2/3)

SEMÁFOROS

Variable semáforo: *S*
Boleano: 0 ó 1 (como los cierres)
General: Entero >= 0

Funciones:
Wait(S) o P(S)
Si *S* == 0 entonces espera a que *S* > 0
Si *S* > 1 entonces *S* = *S* - *1*

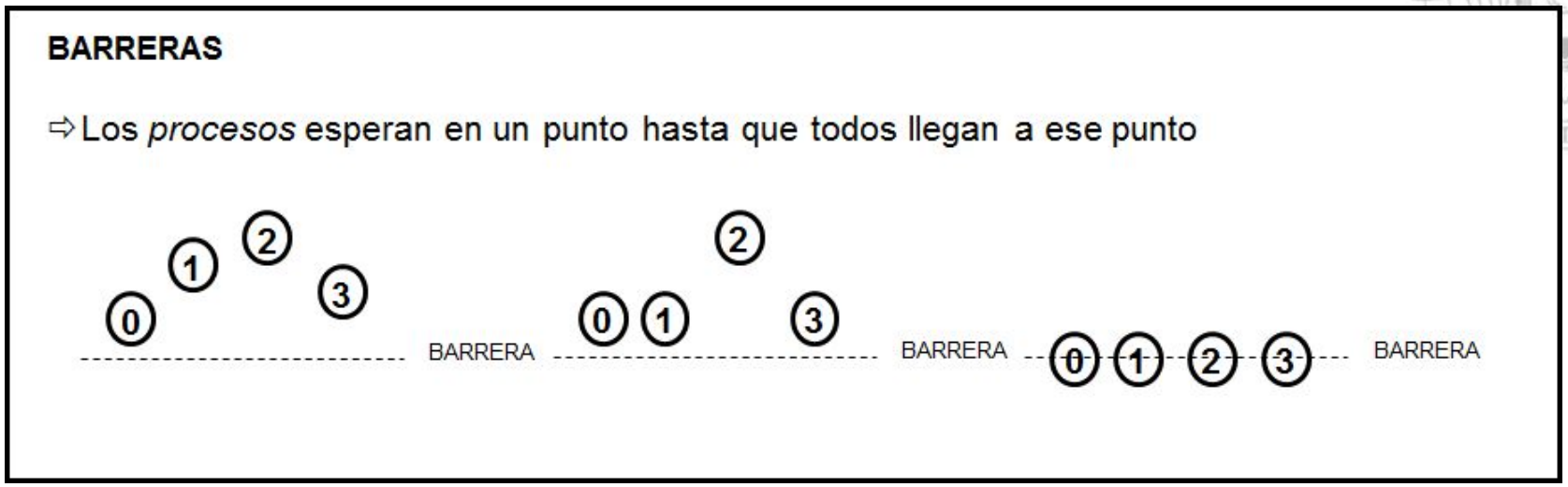
Signal(*S*) o V(S):
S = *S* + *1*

```
P1:   P(S)
      Sección crítica de P1
      v(S)
```

```
P2:   P(S)
      Sección crítica de P2
      v(S)
```

Semáforos generales:
- Si se inicializa con *M*, puede haber *M* procesos en su sección crítica simultanamente
- Problemas productores-consumidores o lectores-escritores

Funciones de Sincronización de Alto Nivel (3/3)



- Otras funciones del API (lenguaje de alto nivel +. . .)
  - Crear procesos
  - Destruir procesos
  - Identificar procesos

Procesos: fork() crea copia del padre con nuevo espacio de memoria virtual

Ventajas e Inconvenientes (1/2)

Concurrencia:	Dos o más threads se pueden ejecutar simultáneamente Esta alternativa incrementa la eficiencia incluso en un solo procesador
Paralelismo:	Hay concurrencia en máquinas multiprocesador El paralelismo surge de modo automático en sistemas multiprocesador

- Mejora de la respuesta de las aplicaciones
  - Algunas aplicaciones cortan la interacción con el usuario cuando realizan una función
  - La programación con threads permite que cada función se realice por un thread de modo independiente
- Uso de sistemas multiprocesador de modo más eficiente
  - Una aplicación paralelizada con threads no tiene que tener en cuenta el número de procesos disponibles
  - El rendimiento mejora automáticamente al añadir más procesos

Ventajas e Inconvenientes (2/2)

- Mejora la estructura de los códigos
  - Muchos programas se estructuran de modo más lógico como unidades independientes
- Uso de menos recursos del sistema
  - Los sistemas UNIX clásicos soportan el concepto de thread (1 proceso = 1 thread)
  - Sin embargo, el mantenimiento de procesos es muchos más costoso que el de threads
- Mejora de rendimiento
  - En un multiprocesador los programas se ejecutan en menor tiempo de ejecución
  - En monoprocesador el tiempo de respuesta es menor, al poder solapar acciones (cómputo con E/S)

Procesos versus Threads (1/3)

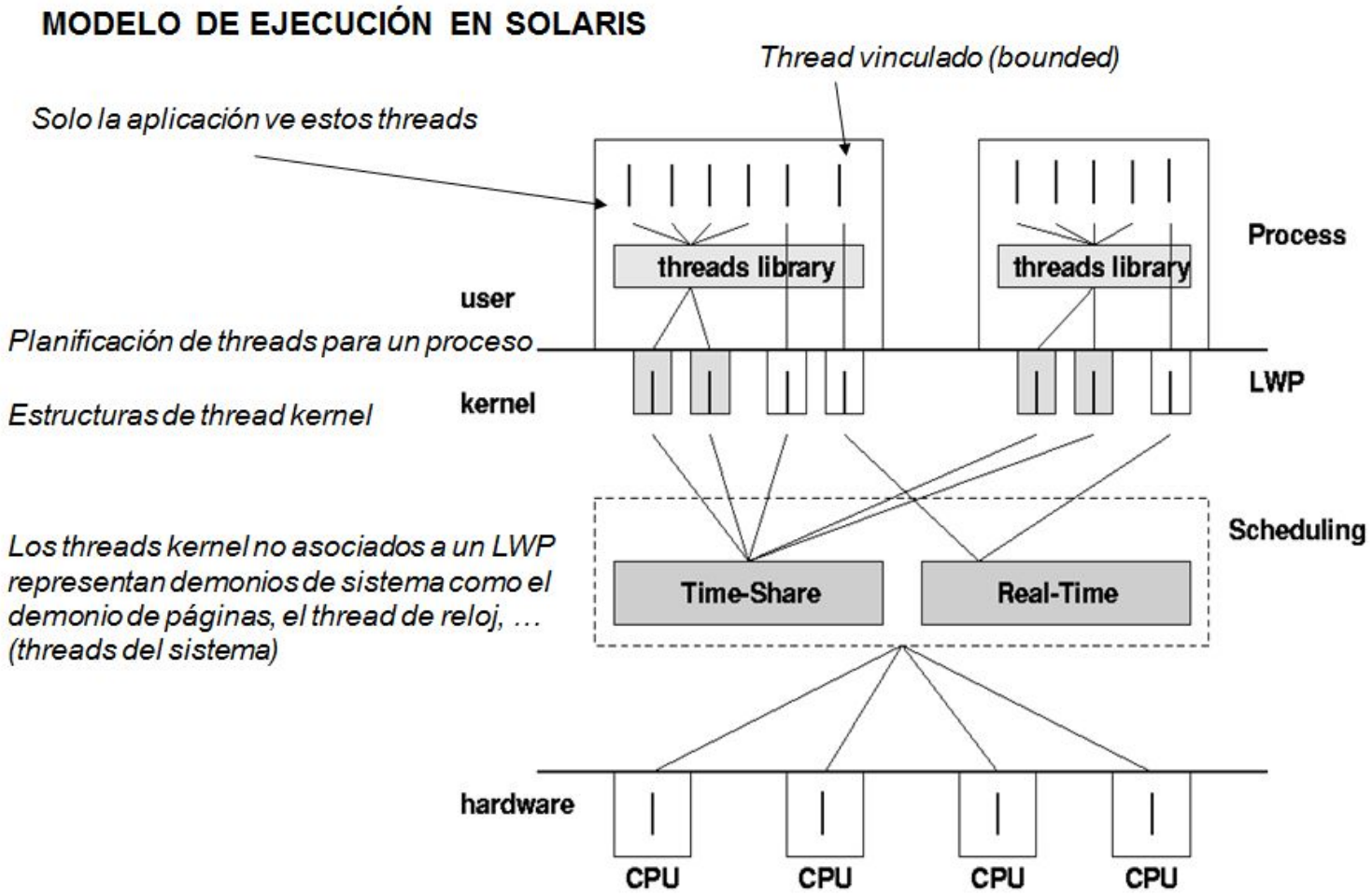
Procesos

- Espacio de direcciones con uno o más threads de control
- Se caracterizan por un contexto software y hardware
- La sobrecarga por creación o carga de proceso es muy alta
- Hay mecanismos artificiales para que varios procesos compartan alguna página de memoria
- Los procesos se diseñaron teniendo en cuenta la protección

Threads

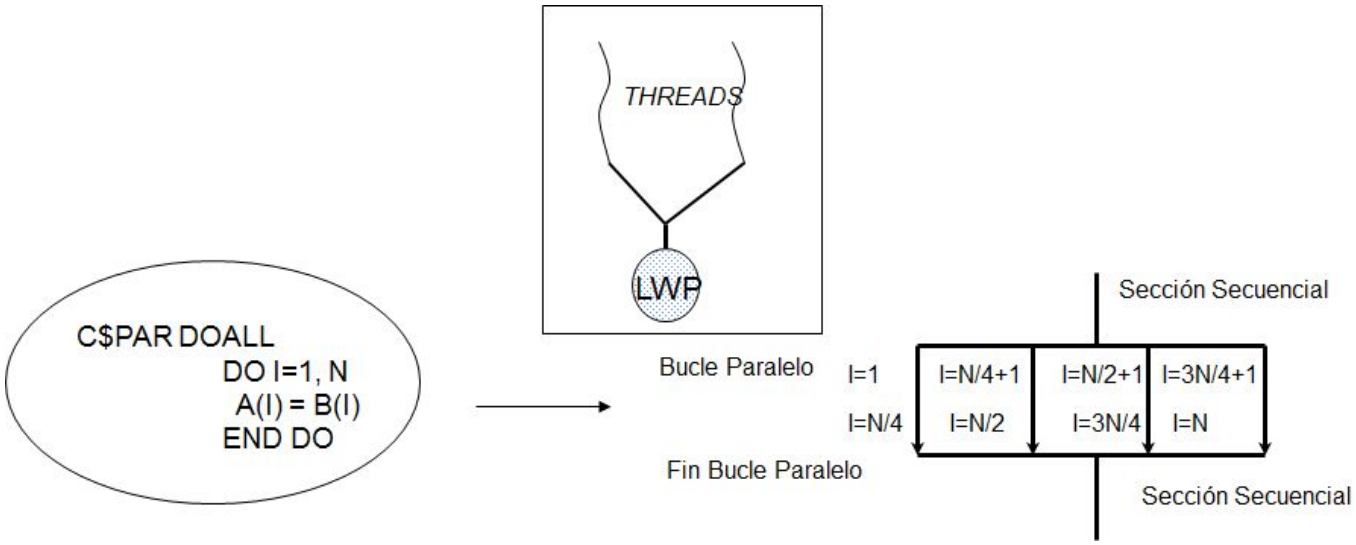
- Flujo de control dentro de un proceso, que coopera con otro thread para resolver un problema
- Los threads comparten un único espacio de direcciones (comunicación sencilla y eficiente)
- Se caracterizan por un contexto hardware
- La sobrecarga por la creación o cambio de thread es muy pequeña
- Los threads comparten memoria de modo natural
- Un thread puede afectar a otro

Procesos versus Threads (2/3)



Procesos versus Threads (3/3)

Threads de aplicación



- La gestión de los threads es responsabilidad del programador
- Cada thread tiene su propia pila
- Todos los threads comparten la memoria del proceso y el resto de contexto software (ficheros abiertos, directorio actual, manejadores de señales, …)
- Los threads se ejecutan de modo independiente (y concurrente)

Ejemplos de Sistemas con Programación por Medio de Threads

Librerías de threads y APIs

- La historia comienza en 1960 y dentro de UNIX en 1980
- Todas soportan funciones muy semejantes
- Sin embargo, hay diferencia en la API de las librerías de los diferentes fabricantes
  - DEC Alpha
  - OS/...
  - Windows ...

Programación por medio de threads

Posix Threads

```
void *print_message_function( void *ptr );
pthread_mutex_t mutex;
void main() {
  pthread_t thread1, thread2;
  pthread_attr_t pthread_attr_default;
  pthread_mutexattr_t pthread_mutexattr_default;
  struct timespec delay;
  char *message1 = "Hello";
  char *message2 = "World\n";

  delay.tv_sec = 10;
  delay.tv_nsec = 0;

  pthread_attr_init(&pthread_attr_default);
  pthread_mutexattr_init(&pthread_mutexattr_default);

  pthread_mutex_init(&mutex, &pthread_mutexattr_default);
  pthread_mutex_lock(&mutex);

  pthread_create(&thread1, &pthread_attr_default,
                 (void*)print_message_function, (void*) message1);
  pthread_mutex_lock(&mutex);
  pthread_create(&thread2, &pthread_attr_default,
                 (void*)print_message_function, (void*) message2);
  pthread_mutex_lock(&mutex);
  exit(0);
}
```

```
void *print_message_function( void *ptr )
{
      char *message;
      message = (char *) ptr;
      printf("%s ", message);
      pthread_mutex_unlock(&mutex);
      pthread_exit(0);
}
```