UNIVERSIDAD CARLOS III DE MADRID
DEPARTAMENTO DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA. ESTRUCTURA DE COMPUTADORES
10 de junio de 2005

Fecha de publicación de las notas: 17 de junio de 2005
Fecha de revisión: 20 de junio 2005 a las 12:00 en el despacho 2.2 A08.

Para la realización del presente examen se dispondrá de **dos horas y media. NO** se podrán utilizar libros ni apuntes. Entregar cada ejercicio en hojas separadas.

Problema 1. 1 punto

1. Describe los mecanismos *test&set* y *test&test&set* para una arquitectura de memoria compartida basada en bus en **términos de tráfico de bus**.
2. Explica qué es la **latencia** de comunicación así como los factores que la componen.

Problema 2. 1.5 puntos

El siguiente programa paralelo se ejecuta en 3 procesadores. Sus variables están declaradas en un arquitectura de memoria compartida e inicializadas a 0 (`x=0, y=0`).

Procesador 1	**Procesador 2**	**Procesador 3**
(1a) x=1	**(2a) print x** **(2b) print y**	**(3a) y=1** **(3b) print x**

Se pide:

1) Explica brevemente qué es la consistencia secuencial.

2) De las siguientes combinaciones de salida, indica cuales son posibles bajo una arquitectura que mantienen una coherencia secuencial. **Justifica tu respuesta diciendo la secuencia de instrucciones en el caso de una solución válida y por qué, en caso de no ser válida, no se puede producir.**

print x (procesador 2)	print y	print x (procesador 3)
0	0	0
0	0	1
0	1	1
1	0	0
1	1	0
1	1	1

Problema 3. 1.5 puntos

Las siguientes dos secciones de código son ejecutadas por diferentes procesadores en un sistema que mantiene una consistencia secuencial.
Si asumimos que la variable i está inicializada a valor cero (0), se pide indicar los posibles valores que se pueden imprimir, justificando, en cada caso, el orden de ejecución que los produce.

```
PROCESADOR 1

i=5

lock(l):
     print  i
unlock(l)
```

```
PROCESADOR 2


lock(l):
          for j=1 to 10
               i=i+1
          end
unlock(l)
```

Problema 4. 2.5 puntos

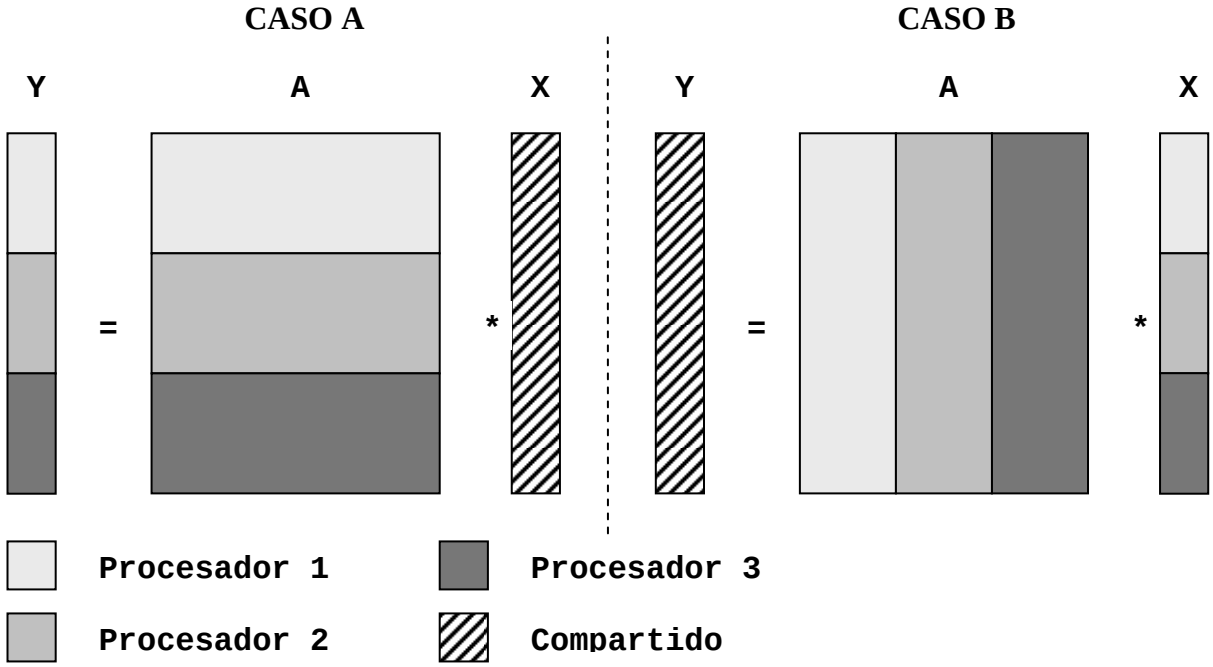
El siguiente fragmento de código realiza el producto de la matriz A por el vector X y almacena el resultado en el vector Y. Asumimos que todas las entradas de Y han sido previamente inicializadas a cero.

```
for ( i = 0; i < n; i++)   % Lazo A
{
   for ( j = 0; j < n; j++)  % Lazo B
   {
     Y[i] = A[i][j] * X[j];
   }
}
```

Se proponen los siguientes dos esquemas de paralelización sobre una arquitectura UMA con un esquema de coherencia caché con una **política de invalidación** (*write-invalidate*).

- CASO A: el lazo A es el único paralelo y las iteraciones están repartidas por bloques.
- CASO B: el lazo B es el único paralelo y las iteraciones están repartidas por bloques.

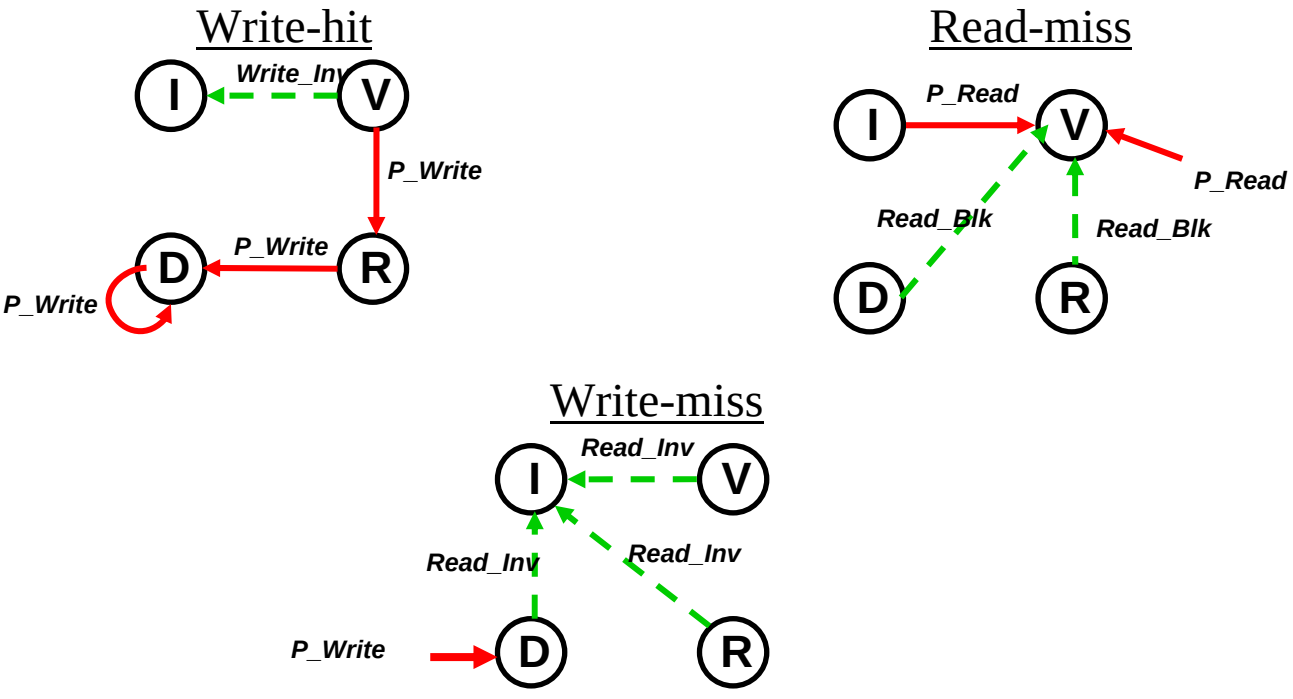
Las figuras siguientes muestran cómo son los datos accedidos por cada procesador:



Por simplificación, vamos a asumir que cada línea caché sólo puede contener una entrada de Y, A o X. Bajo esta suposición indique:

1. Cuántas invalidaciones sufrirá en cada caso una determinada línea caché que contiene una entrada de Y en función del número de filas (n) y número de columnas (n) de la matriz y del número de procesadores (NP). En su respuesta indique el mejor y el peor caso en términos de rendimiento.
2. Razonar el tamaño de grano existente en cada caso.
3. **Únicamente para el Caso B** y utilizando el **protocolo *write-once***, enumera los estados posibles que pueden tener las siguientes líneas de caché asumiendo que inicialmente todas ellas tienen un estado INVALID.
    a. Una línea con una entrada genérica de Y.
    b. Una línea con una entrada genérica de A.
    c.

El siguiente esquema contiene algunas transiciones en el diagrama de estados del protocolo *write-once.*



Problema 5. 1.5 puntos

El siguiente fragmento de código es una variación del código anterior en **donde los valores se acumulan sobre Y y el índice "*j*"del lazo B comienza con valor "*i*"**. Asumimos que el código se ejecuta en un sistema de memoria compartida para el que una operación aritmética (suma y multiplicación) consume *1ns* en ejecutarse y que los accesos a memoria se realizan sin ningún coste (latencia de 0*ns*).
El código está paralelizado empleando las siguientes directivas de open-MP

```
#pragma omp parallel shared(A,X) private(j, i)
#pragma omp for schedule(i,BLOCK)
for ( i = 0; i < n; i++)   % Lazo A
{
   for ( j = i; j < n; j++)  % Lazo B
   {
     Y[i] = Y[i] + A[i][j] * X[j];
   }
}
```

Se pide:

1) Indicar el tiempo de ejecución del **algoritmo secuencial** en función del parámetro "*n*" e ignorando el coste de los lazos (incrementar índices "*i*" y "*j*").

2) Indicar el máximo *speedup* que se puede alcanzar en función de "*n*" y del número de procesadores *NP*.

Problema 6. 2 puntos

Suponga tener un *buffer* circular en el que un proceso productor deposita información y del que un proceso consumidor retira información. Implemente en C, utilizando *threads*, las partes de código del programa anexo señaladas con:

a. Acciones 1 del Productor
b. Acciones 2 del Productor
c. Acciones 1 del Consumidor
d. Acciones 2 del Consumidor

```
#define MAX_BUFFER      1024            /* tamaño del buffer */
#define DATOS_A_PRODUCIR 100000   /* datos a producir */
pthread_mutex_t mutex;      /* mutex para controlar el acceso al buffer compartido */
pthread_cond_t no_lleno;                /* llenado del buffer */
pthread_cond_t no_vacio;                /* vaciado del buffer */
int n_elementos;                         /* elementos en el buffer */
int buffer[MAX_BUFFER];                 /* buffer común */
main(int argc, char *argv[])
{       pthread_t th1, th2;

        pthread_mutex_init(&mutex, NULL);
        pthread_cond_init(&no_lleno, NULL);
        pthread_cond_init(&no_vacio, NULL);
```

```
        pthread_create(&th1, NULL, Productor, NULL);
        pthread_create(&th2, NULL, Consumidor, NULL);

        pthread_join(th1, NULL);
        pthread_join(th2, NULL);

        pthread_mutex_destroy(&mutex);
        pthread_cond_destroy(&no_lleno);
        pthread_cond_destroy(&no_vacio);
        exit(0);
}


void Productor(void) {  /* código del productor */
   int dato, i ,pos = 0;
   for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
      dato = i;                                          /* producir dato */

⇒       /* Acciones 1 del Productor */

      buffer[pos] = dato;
      pos = (pos + 1) % MAX_BUFFER;
      n_elementos ++;

⇒       /* Acciones 2 del Productor */

   }
   pthread_exit(0);
}

void Consumidor(void) {                              /* código del consumidor */
   int dato, i ,pos = 0;
   for(i=0; i < DATOS_A_PRODUCIR; i++ ) {

⇒       /* Acciones 1 del Consumidor */

      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      n_elementos --;

⇒       /* Acciones 2 del Consumidor */

      printf("Consume %d \n", dato);                 /* consume dato     */
   }
   pthread_exit(0);
}
```

Nota:

```
int pthread_mutex_lock    (pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_cond_wait     (pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal   (pthread_cond_t *cond);
```