REPASO PARCIAL 1

(1 punto) 1. Dado el siguiente programa concurrente:

```
static class Hilos {
  static class MiHilo extends Thread {
    int n = 0;
    public void run () {
      for(int i=0; i<100; i++)
        n++;
    }
  }
```

```
  public void main(String[] args) {
    Thread t = new MiHilo();
    t.start(); ← arranca el thread
    t.run();
    hacerAlgo(); // hace algo...
    try {t1.join();}
    catch(InterruptedException e){}
    System.out.println(t.n);
  }
}
```

¿Cuál será la salida por consola al ejecutar el main? Se pide marcar la afirmación correcta.
(a) 200
(b) No se puede saber porque podría haber condiciones de carrera.
(c) 100

volatile → se actualiza el valor de n inmediatamente (cuando se hace una operación)
static → variable compartida (se crea la primera vez)
static → en una clase no puede haber clases hijas y en un método no se puede sobrescribir
t.start() → arranca un nuevo hilo (proceso ligero)
t.run() → llamamos al método run() de la clase MiHilo

Si tuviéramos t.run(); t.start(); entonces el resultado sería 200 (no se produce condición de carrera, es decir, hay exclusión mutua).
En el caso de tener t.start(); t.run(); SÍ que se puede producir condición de carrera, ya que se puede acceder simultáneamente al método run() por dos sitios diferentes.

(1 punto) 2. Dado el siguiente programa concurrente:

```
static class Hilos {
  static class MiHilo extends Thread {
    int n = 0;
    public void run () {
      for(int i=0; i<100; i++)
        n++;
    }
  }

  public void main(String[] args) {
    Thread t = new MiHilo();
    t.start(); ← SI arranca proceso
    t.run(); ← NO arranca proceso
    hacerAlgo(); // hace algo...
    try {t1.join();}
    catch(InterruptedException e){}
    System.out.println(t.n);
  }
}
```

¿Cuál será la salida por consola al ejecutar el main? Se pide marcar la afirmación correcta.
(a) El número máximo de procesos ejecutando a la vez será 2 y el método main podría terminar antes que el thread t.
(b) El número máximo de procesos ejecutando a la vez será 3 y el método main terminará siempre después que el thread t.
(c) El número máximo de procesos ejecutando a la vez será 2 y el método main terminará siempre después que el thread t.

Tenemos el proceso principal + el hilo t
El main siempre terminará después que el thread, ya que tenemos un t.join(), que se espera a que termine el hilo

(1 punto) 3. Dada la siguiente implementación de una solución al problema de la exclusión mutua con espera activa:

```
static volatile boolean inc_quiere = false;
static volatile boolean dec_quiere = false;
static volatile int cont = 0;
```

```
class Incrementador extends Thread {
  public void run() {
    for(int i=0; i<N_OPS; i++) {
      inc_quiere = true; (1)
      while (dec_quiere) {} (3)
      cont++;
      inc_quiere = false;
    }
  }
}
```

```
class Decrementador extends Thread {
  public void run() {
    for (int i=0; i<N_OPS; i++) {
      dec_quiere = true; (2)
      while (inc_quiere) {} (4)
      cont--;
      dec_quiere = false;
    }
  }
}
```

Suponiendo que tenemos un proceso de tipo Incrementador y otro proceso Decrementador, se pide marcar la afirmación correcta.
(a) El programa no garantiza la ausencia de esperas innecesarias.
(b) El programa no garantiza la exclusión mutua en el acceso a la sección crítica (`cont++` y `cont--`).
(c) El programa no garantiza la propiedad de ausencia de interbloqueo.

Se quedan los dos esperando eternamente en los while's sin avanzar → interbloqueo

(1.5 puntos) 4. Dado el siguiente programa:

```
class Carreras {
  private static volatile int a = 0;
  static class Hilo extends Thread {
    private static volatile int b = 0;
    private volatile int c = 0;
    public void run() {
      a++;
      b++;
      c++;
      System.out.print(c);
    } Imprime el valor de c
  }
  public static void main
     (final String[] args) throws Exception {
    Thread t1 = new Hilo();
    Thread t2 = new Hilo();
    t1.start();
    t2.start();
    System.out.print("ejecutando");
    t1.join();
    t2.join();
  }
}
```

Se pide marcar la afirmación correcta:
(a) La sentencia a++; no es una condición de carrera
(b) La sentencia b++; no es una condición de carrera
(c) La sentencia c++; no es una condición de carrera

a → es común a toda la clase Carreras (tanto a la clase Hilo, como al main()), se actualiza inmediatamente
b → se inicializa cuando hacemos t1 = new Hilo() y con t2 = new Hilo() ya está inicializada porque es compartida
c → NO es compartida → t1 tendrá su propia c y t2 tendrá su propia c

con t1.start() → a=1, b=1, c=1
con t2.start() → a=2, b=2, c=1

(1 punto) 5. Dado el programa de la pregunta anterior. Se pide marcar la afirmación correcta:
(a) "ejecutando11" es una salida posible del programa
(b) "ejecutando11" no es una salida posible del programa

Tenemos 3 posibilidades: 11ejecutando, 1ejecutando1 y ejecutando11

(1 punto) 6. Dado un programa concurrente en el que tres threads instancias de las clases A, B y C comparten las variables x, y, sx y sy.

```
static int x = 1;
static int y = 2;
static Semaphore sx = new Semaphore(0);
static Semaphore sy = new Semaphore(0);
```

```
class A extends Thread {
  public void run() {
    x = x + 1;
    sx.signal();
  }
}
```

```
class B extends Thread {
  public void run() {
    y = y + 1;
    sy.signal();
  }
}
```

```
class C extends Thread {
  public void run() {
    sx.await();
    sy.await();
    System.out.print(x+y);
  }
}
```

Se pide marcar la afirmación correcta:
(a) "5" es una salida posible del programa (si x e y fueran volatile → SERÍA LA ÚNICA SOLUCIÓN POSIBLE)
(b) "5" no es una salida posible del programa

Como no son **volatiles**, entonces puede que al ejecutar los threads A (x = 2) y B (y = 3), no se actualicen antes de que se haga la impresión
Con A → sx.cont = 1; x = 2
Con B → sy.cont = 1; y = 3
Con C → sx.cont = 0, sy.cont = 0 → podría llegar a imprimir 5 si se hubieran actualizado

(1 punto) 7. Si en el código anterior el semáforo sx se inicializa a 1. Se pide marcar la afirmación correcta:
(a) "5" es una salida posible del programa
(b) "5" no es una salida posible del programa

En este caso tenemos también tres posibles soluciones: 3 si no se ha actualizado la memoria, 4 si hemos hecho B y se ha actualizado, y 5 si hubiéramos hecho A y B antes que C y se hubiera actualizado.

(1.5 puntos) 8. Se desea modelar con semáforos una barrera para hilos con la cual se bloquearán todos los hilos hasta que haya llegado el último, momento en el cual permitiremos seguir a todos los hilos.

```
static class Hilos {
  static final int MAX_HILOS = 5;
  static class MiHilo extends Thread {
    static volatile int cont = 0;
    static Semaphore mutex = new
Semaphore(1);
    static Semaphore barrera = new
Semaphore(0);
    public void run() {
      tarea1();
      mutex.await();
      cont = cont + 1;
      if(cont == MAX_HILOS)
        barrera.signal();
      barrera.await();
      barrera.signal();
      mutex.signal();
     tarea2();
    }
  }
}
```

Supongamos que lanzamos MAX_HILOS hilos instancias de MiHilo. Asumiendo que los métodos tarea1() y tarea2() siempre terminan, se pide marcar la afirmación correcta.

(a) El programa implementa siempre la barrera tal como se pretendía.

(b) El programa siempre acabará en interbloqueo

(c) El programa se comporta como una barrera solo en algunas ejecuciones, dependiendo de las velocidades relativas de los procesos.

INICIO:
cont = 0
mutex.cont = 1
barrera.cont = 0

Hilo1	Hilo2	Hilo3	Hilo4	Hilo5
mutex.cont = 0	mutex → bloq	mutex → bloq	mutex → bloq	mutex → bloq
cont = 1				
barrera → bloq				

CASO 2 → inicializamos mutex con MAX_HILOS
cont = 0
mutex.cont = 5
barrera.cont = 0

Hilo1	Hilo2	Hilo3	Hilo4	Hilo5
mutex.cont = 4	mutex.cont = 3	mutex.cont = 2	mutex.cont = 1	mutex.cont = 0
cont = 1	cont = 2	cont = 3	cont = 4	cont = 5
				barrera.cont = 1
barrera → bloq se desbloquea (1)	barrera → bloq se desbloquea (2)	barrera → bloq se desbloquea (3)	barrera → bloq se desbloquea (4)	barrera → bloq se desbloquea (5)

En este caso NO HAY INTERBLOQUEO, PERO SE PRODUCE COND. CARRERA, es decir,
NO HAY EXCLUSIÓN MUTUA

CASO 3 → cambiamos el mutex.signal() de orden en el código
La solución óptima y que respeta el enunciado sería:

```
public void run() {
        tarea1();
        mutex.await();
        cont = cont + 1;
        if(cont == MAX_HILOS)
          barrera.signal();
        mutex.signal();
        barrera.await();
        barrera.signal();
       tarea2();
}
```

cont = 0
mutex.cont = 1
barrera.cont = 0

Hilo1	Hilo2	Hilo3	Hilo4	Hilo5
mutex.cont = 0	mutex.cont = 0	mutex.cont = 0	mutex.cont = 0	mutex.cont = 0
cont = 1	cont = 2	cont = 3	cont = 4	cont = 5
mutex.cont = 1	mutex.cont = 1	mutex.cont = 1	mutex.cont = 1	mutex.cont = 1
				barrera.cont = 1
barrera → bloq se desbloquea (1)	barrera → bloq se desbloquea (2)	barrera → bloq se desbloquea (3)	barrera → bloq se desbloquea (4)	barrera → bloq se desbloquea (5) mutex.cont = 2

(1 punto) 9. Dado el siguiente programa concurrente:

```
static int x = 0; ← COMPARTIDA
static class T extends Thread {
  private int y;
  public T (int y) {
    this.y = y;
  }
  public void run() {
    int z = y;
    z = z + y;
    x = x + z;
  }
}
// Programa principal
Thread[] t = new Thread[] {new T(1), new T(2)};
t[0].start(); t[1].start(); t[0].join(); t[1].join();
```

Se pide marcar la afirmación correcta.

(a) Es necesario asegurar exclusión mutua en el acceso a la variable z.

(b) Ninguna de las otras respuestas es correcta.

(c) Es necesario asegurar exclusión mutua en el acceso a la variable x.

(d) Es necesario asegurar exclusión mutua en el acceso al atributo y.

(1 punto) 10. La clase `PorTurno` implementa un protocolo de acceso a una sección crítica:

```
static int turno = 0; ← COMPARTIDA

static class PorTurno extends Thread {
  private int pid; ← NO COMPARTIDO

  public PorTurno(int pid) {
    this.pid = pid;
  }
  public void run() {
    while (true) {
      s();
      while (turno != pid) {}
      seccionCritica();
      turno = (turno + 1) % MAX_THREADS;
    }
  }
}
```

Dado un programa concurrente con `MAX_THREADS` threads de la clase `PorTurno` compartiendo una variable turno inicializada a 0 y cada una de ellas con un índice pid distinto entre `0` y `MAX_THREADS-1`. Se pide marcar la afirmación correcta.

(a) Garantizada la terminación de `s()` y `seccionCritica()`, el programa no cumple la propiedad de ausencia de inanición. ← el tiempo de espera está acotado (todos acaban SIEMPRE)

(b) Garantizada la terminación de `s()` y `seccionCritica()`, el programa cumple las propiedades de exclusión mutua en `seccionCritica()`, ausencia de interbloqueo y ausencia de inanición pero un proceso podría esperar para ejecutar `seccionCritica()` sin que los demás ejecuten `seccionCritica()` ni compitan por hacerlo.

(c) Garantizada la terminación de `s()` y `seccionCritica()`, el programa no cumple la propiedad de ausencia de interbloqueo. ← sí se cumple ya que en algún punto llegará el pid = 0

(d) El programa no cumple la propiedad de exclusión mutua en `seccionCritica()`. ← NO tenemos dos procesos accediendo simultáneamente a su sección crítica

INTERBLOQUEO → el proceso se queda bloqueado eternamente en un punto del programa SIN QUE SU VALOR CAMBIE NUNCA y no puede continuar

DEADLOCK → el proceso se queda bloqueado eternamente en un punto del programa AUNQUE SU VALOR CAMBIA CONTINUAMENTE (SIN ACERCARSE A LA CONDICIÓN DE SALIDA DEL BUCLE) y no puede continuar

ESPERAS INNECESARIAS → un proceso está bloqueado esperando otro SIN NECESIDAD DE HACERLO, porque no hay nadie ejecutando ninguna sección crítica en ese punto (momento)

INANICIÓN → el tiempo de espera TOTAL del programa está acotado

EXCLUSIÓN MUTUA → si hay exc. mutua, entonces NO se producen cond. carrera, es decir, NO hay dos procesos diferentes intentando acceder a una misma variable compartida. (son inversamente proporcionales)

(2 puntos) 11. La instrucción TST (*Test and set*) es típica de algunas arquitecturas. Su comportamiento se basa en la existencia de una variable c común a varios procesos. Al ejecutar x = TST();, donde x debería ser una variable local al proceso, se puede asumir que se realiza automáticamente la siguiente ejecución: x = c; c = 1. El siguiente programa concurrente hace uso de dicha instrucción para regular el acceso a una sección crítica:

```
public static final void
    main (final String[] args) {

  Thread t1, t2;
  t1 = new T();
  t2 = new T();
  t1.start();
  t2.start();

}
```

```
static class T extends Thread {
  public T() { }
  public void run() {
    int x;
    while(true) {
      Sec_No_Critica();
      do { x = TST(); } while (x != 0);
      Sec_Critica();
      c = 0;
    }
  }
}
```

Se pide marcar la afirmación correcta:
(a) No se garantiza la propiedad de exclusión mutua.
(b) Se puede producir interbloqueo
(c) Puede producirse inanición de un proceso.
(d) Se garantiza la exclusión mutua y la ausencia de inanición sin que haya posibilidad de interbloqueo.

En un primer momento, los dos threads ejecujando el do-while "simultáneamente" y ponen x = 0, por lo que vuelven a ejecutar por segunda vez ambos el bucle do-while. En este punto, c = 1, por lo que ambos ponen x = 1, se salen los dos del bucle y ejecutan simultáneamente su sección crítica → NO HAY EXCLUSIÓN MUTUA (se produce condición de carrera)

(1 punto) 12. Asumiendo que la variable t contiene una referencia a un primer thread que ya ha terminado y que un segundo thread ejecuta t.join(), se pide señalar la respuesta correcta.
(a) El segundo thread se parará en la ejecución de t.join() hasta que el thread referenciado por t le envíe una señal.
(b) El segundo thread no se parará en la ejecución de t.join().

Si el thread en cuestión ya ha terminado, entonces el programa no se queda bloqueado, simplemente continúa (ya que el join() solo espera que el thread al que afecta termine, pero NO es bloqueante).

Lo contrario pasa con los semáforos que SÍ son bloqueantes (el método await() / acquire() bloquea; el método signal() / release() DESbloquea - en caso de que algo estuviera bloqueado previamente)

(1 punto) 3. Supongamos un programa concurrente con procesos (al menos uno) que ejecutan repetidamente operaciones $r.reintegro(x)$ y procesos (al menos uno) que ejecutan repetidamente operaciones $r.ingreso(y)$, siendo r un recurso compartido del tipo especificado a continuación:

C-TAD CuentaBancaria
OPERACIONES
ACCIÓN reintegro: $\mathbb{N}[e]$
ACCIÓN ingreso: $\mathbb{N}[e]$
SEMÁNTICA
DOMINIO:
TIPO: *CuentaBancaria* = $\mathbb{N}$
INICIAL: self = 0

CPRE: $c \leq$ self
reintegro(c)
POST: self = self$^{pre} - c$

CPRE: Cierto
ingreso(n)
POST: self $- n =$ self^{pre}

Se pide señalar la respuesta correcta.

(a) El programa cumple la propiedad de ausencia de interbloqueo.

(b) El programa no cumple la propiedad de ausencia de interbloqueo.

INICIAL → (0) - ingreso(10) -> (10) -ingreso(2)-> (12)
<- reintegro(10)-

COMO LOS INGRESOS SE HACEN REPETIDAMENTE → En algún momento podré hacer un reintegro de una cantidad (por muy grande que sea) → NO HAY INTERBLOQUEO

(1 punto) 4. Dado el programa concurrente descrito en la pregunta 3. **Se pide** señalar la respuesta correcta.

(a) La especificación de la operación de *ingreso* es incorrecta.

(b) La especificación de la operación de *ingreso* es correcta.

Si despejamos, nos queda la operación igual a self = selfPRE + n, es decir, que el ingreso se hace correctamente → La especificación es CORRECTA

(1½ puntos) 5. Obsérvese la siguiente implementación del recurso compartido CuentaBancaria especificado en la pregunta 3:

```
class CuentaBancaria {
   private Semaphore saldo = new Semaphore(0);
   private Semaphore atomic = new Semaphore(1);

   public void reintegro(int c) {
      atomic.await();
      for (int i = 0; i < c; i++)
         saldo.await();
      atomic.signal();
   }

   public void ingreso(int c) {
      for (int i = 0; i < c; i++)
         saldo.signal();
   }
}
```

La idea principal consiste en que el semáforo `saldo` represente el valor interno (de tipo $\mathbb{N}$) del recurso. Asumiendo que se quiere atender a los procesos que quieren realizar reintegros en estricto orden de llegada, se pide señalar la respuesta correcta.

(a) Es una implementación correcta del recurso compartido.

(b) Es una implementación incorrecta del recurso compartido.

INICIAL:
saldo.cont = 0
atomic.cont = 1

reintegro(4) → atomic.cont = 0, se bloquea en saldo.await()

ingreso(3) → saldo.cont = 3 → desbloquea 3 veces la operación de reintegro anterior, pero saldo.cont = 0 y reintegro sigue esperando 1

reintegro(1) → se queda bloqueado en atomic.await() hasta que la operación de reintegro anterior no haya terminado

Luego, las operaciones de reintegro se ejecutan en ESTRICTO orden de llegada → Es una implementación correcta del recurso compartido

(1.5 puntos) 15. Obsérvese la siguiente implementación del recurso compartido CuentaBancaria:

```
class CuentaBancaria {
  static Semaphore saldo = new Semaphore(0);
  static Semaphore atomic = new Semaphore(1);

   public void reintegro(int c) {
     atomic.await();
     for(int i=0; i<c; i++)
       saldo.await();
     atomic.signal();
   }

   public void ingreso(int c) {
     atomic.await();
     for(int i=0; i<c; i++)
       saldo.signal();
   }
}
```

La idea principal consiste en que el semáforo saldo representa el valor interno (de tipo N) del recurso. Asumiendo que se quiere atender a los procesos que quieren realizar reintegros en estricto orden de llegada, se pide señalar la respuesta correcta:

(a) Es una implementación incorrecta del recurso compartido

(b) Es una implementación correcta del recurso compartido

ingreso(2) → atomic.cont = 0, saldo.cont = 2

reintegro(2) → se queda bloqueado en atomic.await() Y NUNCA VA A PERMITIR AVANZAR YA MÁS EL PROGRAMA

ingreso(1) → también se queda bloqueado en atomic.await()

(1 punto) 16. Se pide señalar la respuesta correcta teniendo en cuenta que no hay otros métodos más allá del método run:

(a) El acceso a un atributo no estático y privado de tipo int de un thread desde un método run nunca es una sección crítica.

(b) El acceso a un atributo no estático y privado de tipo int de un thread desde un método run puede ser una sección crítica.

¿Y si tenemos el siguiente caso?

Thread t = new Thread();

t.start(); ← arranca proceso y accede al método run()

t.run(); ← accede al método run() (puede que a la vez, por lo que sería una sección crítica)