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();
    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

(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();
    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) 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.
(b) El número máximo de procesos ejecutando a la vez será 2 y el método main terminará siempre después que el thread t.

(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;
      while (dec_quiere) {}
      cont++;
      inc_quiere = false;
    }
  }
}
```

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

(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);
    }
  }
```

```
  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

(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

(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
(b) "5" no es una salida posible del programa

(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

(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.wait();
      cont = cont + 1;
      if(cont == MAX)
        barrera.signal();
      barrera.wait();
      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.

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

```
static int x = 0;
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;

static class PorTurno extends Thread {
  private int pid;

  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.
(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.
(d) El programa no cumple la propiedad de exclusión mutua en `seccionCritica()`.

(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.

(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().

(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.

(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.

(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.

(1 punto) 6. **Se pide** señalar la respuesta correcta[1].

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

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