Universitat Politècnica de Catalunya
Department d'Enginyeria Telemàtica

Escola Tècnica Superior d'Enginyeria de
Telecomunicació de Barcelona

Problemes AST

Josep Cotrina
Marcel Fernandez
Jordi Forga
Juan Luis Gorricho
Francesc Oller

Part 1

Mecanismes d'espera activa

1. Es vol implementar un mètode `rwlock.enter(int mode)` per solucionar el problema Lectors/Escriptors amb espera activa. `rwlock (read/write lock)` encapsula un sencer que val 0 si la taula esta lliure, -1 si esta sent accedida per un escriptor o el nombre de lectors si esta sent accedida per un conjunt de lectors. L'argument `mode` és 1 si s'accedeix per llegir i -1 si s'accedeix per escriure. Es disposa de les operacions `ENTERZC/EXITZC` per protegir amb espera activa l'accés a Zones Crítiques. Realitzar la codificació d'aquest mètode.

2. Si es disposa d'una operació atómica `n.atomic_dec()` que decrementa el valor d'un comptador i retorna el resultat decrementat. Si s'inicialitza el comptador a 1, `n = 1`, com es pot resoldre el problema d'Exclusió Mútua.

3. Es vol implementar la construció

   ```
   await(exp_bool){
       codi;
   }
   ```

   (construcció `espera_a_que`) que bloqueja el procés fins que `exp_bool` sigui certa i llavors executa `codi`. Donar una possible codificació d'aquesta construcció.

4. Dos productors, amb comportament idèntic, emeten en seqüència els números 1 i 2 amb el codi no atòmic

   ```
   ...println(...);
   buffer.put(i);
   ```

   i acaben.

   Dos consumidors llegeixen dos dels números cadascun amb el codi no atòmic

   ```
   ...println(...+buffer.get());
   ```

i acaben.

`put` i `get` són operacions d'un monitor buffer. Quins dels següents entrellaçats són correctes?:

```
Entrellaçat 1:

Consumidor(2) obté 1
Consumidor(1) obté 2
Consumidor(1) obté 1
Consumidor(2) obté 2

Entrellaçat 2:

Consumidor(1) obté 1
Consumidor(2) obté 1
Consumidor(1) obté 2
Consumidor(2) obté 2

Entrellaçat 3:

Consumidor(2) obté 2
Consumidor(1) obté 1
Consumidor(2) obté 1
Consumidor(1) obté 2
```

5. **Algorisme de Peterson.** Es vol encapsular l'algorisme de Peterson en una classe `Mutex` amb mètodes `enter()` i `exit()`.

   Es demana:

   (a) Els camps de dades de la classe `Mutex`.

   (b) La realització del mètode `enter()` suposant que es fa servir una classe `MyThread` que té un camp `id` sencer que l'identifica.

6. Es disposa d'una classe `Counter` amb els mètodes

   - `void dec()` que decrementa atòmicament un sencer i no retorna cap valor.
   - `int intValue()` que retorna atòmicament el valor del sencer.
   - `void setValue(int)` que modifica el valor del sencer segons el paràmetre.

   Se suposa que el sencer pot guardar qualsevol valor sense desbordament.

   Es proposen dues solucions:

```
Solució 1:
    // Init
    Counter n = new Counter(1);
    ...
```

```
    // Enter
    do {
        n.dec();
    } while (n.intValue() != 0) ;
    ...
    // Exit
    n.setValue(1);
```

Solució 2:

```
    // Init
    Counter n = new Counter(0);
    ...
    // Enter
    do {
    n.dec();
    } while (n.intValue() != -1) ;
    ...
    // Exit
    n.setValue(0);
```

Les solucions proposades resolen el problema d'Exclusió Mútua ?

7. Es disposa d'una classe `Counter` amb els mètodes

   - `int dec()` que decrementa atòmicament un sencer i retorna el valor antic.
   - `void setValue(int)` que modifica el valor del sencer segons el paràmetre..

   Se suposa que el sencer pot guardar qualsevol valor sense desbordament.

   Utilitzant la classe `Counter` resoldre el problema d'Exclusió Mútua.

8. Es disposa d'una classe `Counter` amb els mètodes

   - `int dec()` que decrementa atòmicament un sencer i retorna el valor antic.
   - `int inc()` que incrementa atòmicament un sencer i retorna el valor antic.
   - `void setValue(int)` que modifica el valor del sencer segons el paràmetre..

   Se suposa que el sencer pot guardar qualsevol valor sense desbordament.

   Es proposen 3 solucions al problema Lectors/Escriptors, on la taula pot ser accedida per un nombre arbitrari de Lectors o bé per un únic Escriptor.

Solució 1:

```
    // StartRead                // StartWrite
    while (n.inc() == -1) ;     while (n.dec() != 0) ;
    ...                         ...
    // EndRead                  // EndWrite
```

```
    n.dec() ;                   n.setValue(0);
```

```
Solució 2:
    // StartRead                // StartWrite
    while (n.inc() == -1) ;     while (n.dec() != 0) ;
    ...                         ...
    // EndRead                  // EndWrite
    n.setValue(0);              n.inc();
```

```
Solució 3:
    // StartRead                // StartWrite
    while (n.inc() == -1)       while (n.dec() != 0)
    n.dec();                    n.inc();
    ...                         ...
    // EndRead                  // EndWrite
    n.dec() ;                   n.inc();
```

Les solucions proposades resolen el problema Lectors/Escriptors ?

9. **One Lane Bridge.** Per un pont poden circular cotxes, que es modelen com a threads, en un sentit o en l'altre però no en ambdós sentits a la vegada. Per això es demana dissenyar una classe Bridge amb els mètodes void enter(boolean way) i void exit().

   Un cotxe quan vol accedir al pont fa una crida al mètode enter(boolean way), passant com a paràmetre la seva direcció. Si el pont està ocupat per cotxes que circulen en direcció contrària, aquest s'haurà d'aturar. Un cop un cotxe abandona el pont crida el mètode exit().

Part 2

Monitors

1. Sigui un procés productor i `n` processos consumidors que comparteixen un buffer de capacitat `b`. Cada element depositat pel productor ha de ser agafat pels `n` consumidors, però aquests poden agafar els elements en qualsevol moment. Un lector pot anar avançat com a molt `b` posicions respecte la resta de lectors.

   En resoldre el sistema fent servir monitors, la classe `Monitor_Buffer` queda definida com:

```
Monitor_Buffer :

int dades[b];   //vector de dades
int llegits[b]; //nº de lectures sobre la posició
                //del vector dades[]
boolean hi_ha[b];   //posició del vector de
                    //dades[] lliure/ocupada
int punter_lec[n];  //punter de lectura de cada lector
int punter_esc = 0; //punter d'escriptura de l'escriptor
int num_esc = 0;        //nombre d'escriptures total
int num_lec[n] = 0; //nombre de lectures
                    //total de cada lector
condition pot_llegir;
condition pot_escriure;
//inicialització corresponent;
int get(int id) { . . . }; void put( int elem) { . . . };
```

   (a) Donar una possible solució del mètode `int get(int id)`.

   (b) Donar una possible solució del mètode `void put(int elem)`.

2. (One lane bridge) Per un pont poden circular cotxes, que es modelen com a threads, en un sentit o en l'altre però no en ambdós sentits a la vegada. Per això es dissenya un monitor `bridge` amb semàntica Signal and Continue que no te en compte cap criteri de justicia a l'hora d'accedir al pont. El monitor bridge té definits dos mètodes: `enter()` i `exit()` i els atributs:

   `nombre_cotxes`: cotxes que estan passant pel pont en cada moment

`sentit_actual`: sentit en que estan passant els cotxes en cada moment

(a) Implementar el mètode `enter()` en pseudo-codi.

(b) Implementar el mètode `exit()` en pseudo-codi.

3. Es vol implementar el firmware d'un commutador ATM amb un canal de sortida sobre el qual es transmet informació de diferents connexions amb prioritats. Aquest firmware vé definit per diferents processos productors (connexions) i 1 procés consumidor (canal de sortida) que comparteixen un buffer de transmissió de capacitat `b`. Els processos productors tenen definida una prioritat d'accés sobre el buffer de 0 a `n-1`, essent `n-1` la prioritat màxima. Això vol dir que mentre hi hagi espai al buffer poden accedir-hi lliurement, però quan el buffer s'ha omplert, els processos productors seran aturats i posteriorment despertats segons l'ordre de prioritat.

   En resoldre el sistema fent servir monitors amb variables de condició amb semàntica Signal and Continue, la classe `Buffer` queda definida com:

```
import Monitor.*;

public class Buffer extends Monitor {

    private CircularQueue q;

    //variable de condició per aturar el consumidor:
    private Condition empty;

    //variables de condició segons prioritat pels product.:
    private Condition full[];

    //nombre de productors esperant d'una prioritat:
    private int producer[];

    //prioritat més alta en qualsevol moment:
    private int priority;

    public Buffer(int cp){
        q = new CircularQueue(cp);
        empty = create_cond();
        producer = new int[n];
        full = new Condition[n];
        for(int i=0; i<n+1; i++){
            full[i] = create_cond();
            producer[i] = 0;
        }
        priority = -1;
    }

    public Object get() {...}
```

```
    public void put(Object element, int my_priority) {...}
}
```

(a) Donar una possible implementació del mètode `Object get( )`.

(b) Donar una possible implementació del mètode `void put(Object element, int my_priority)`.

4. Sigui un procés escriptor i `n` processos lectors que comparteixen un buffer de capacitat `b`. Cada element dipositat pel escriptor ha de ser agafat pels `n` lectors, però aquests poden agafar els elements en qualsevol moment. Un lector pot anar avançat com a molt `b` posicions respecte la resta de lectors degut a la capacitat limitada del buffer.

   En resoldre el sistema fent servir monitors amb variables de condició amb semàntica Signal and Continue, la classe `Monitor_Buffer` queda definida com:

```
public class Monitor_Buffer extends
Monitor {
    protected int N;
    protected int B;
    // vector de dades:
        protected int[] dades;
    // lectures disponibles de cada lector:
        protected int[] lect_disp;
    // punter d'escriptura (de 0 a B-1):
        protected int punter_escr;
    protected Condition pot_escriure;
    protected Condition pot_llegir;

    public Monitor_Buffer(int n, int b) {
        N = n;
        B = b;
        dades = new int[B];
        lect_disp = new int[N];
        pot_escriure = create_cond();
        pot_llegir = create_cond();
    }

    public void put(int elem) {...}

    // cid és l'identificador del lector(de 0 a N-1):
    public int get(int cid) {...}
}
```

(a) Donar una possible implementació del mètode `void put(int elem)`.

(b) Donar una possible implementació del mètode `int get(int cid)`.
Nota: l'expressió `((punter_escr + B - lect_disp[cid]) % B)` equival a `(punter_escr - lect_disp[cid])` amb aritmètica mòdul `B`.

5. En un sistema hi ha dos tipus de processos concurrents `NoPrior` i `Prior` que accedeixen en exclusió mútua a un recurs comú. Els processos `Prior` tenen prioritat sobre els `NoPrior`, és a dir, quan arriba un procés `Prior` passa davant de tots els `NoPrior` que puguin estar esperant per accedir al recurs.

   Es vol resoldre aquest sistema fent servir monitors de Java. D'aquesta forma es defineix una classe amb els següents atributs i les seves inicialitzacions que gestiona l'accés a través dels mètodes `demanar_recurs( )` i `alliberar_recurs( )`:

```
//Número de processos Prior en espera:
    int cuaP=0;
//Número de processos NoPrior en espera:
    int cuaNoP=0;
//Indica si algun procés està accedint al recurs:
    boolean ocupat=false;
```

   (a) Implementar el mètode `demanar_recurs`. El paràmetre `tipus` representa el tipus de procés (0=`NoPrior` i 1=`Prior`).

   (b) Implementar el mètode `alliberar_recurs`.

6. Considerem el problema Productor/Consumidor amb monitors i ordre de servei FIFO on les operacions son servides en l'ordre d'arribada dels threads (suposant que la cua del bloc `synchronized` té servei FIFO).

   Es demana:

   (a) La programació amb mecanismes de sincronització JAVA del constructor de la clase `bufferFIFO`.

   (b) La programació amb mecanismes de sincronització Java de l'operació `put`

7. Considerem el problema Lectors/Escriptors de manera que es garanteix únicament l'accés concurrent de tots els Lectors i l'accés exclusiu d'un Escriptor.

   Es demana:

   (a) La programació de les operacions `synchronized void StartRead()` i `synchronized void StartWrite()`.

   (b) La programació de la operació `synchronized void EndRead()`.

   (c) La programació de la operació `synchronized void EndWrite()`.

8. Molts sistemes aconsegueixen implementar l'exclusió mútua definida en els paquets de semàfors i monitors programats amb llenguatges d'alt nivell gràcies a l'ús d'una instrucció de llenguatge màquina disponible en molts microprocessadors anomenada TestAndSet. Es tracta d'una instrucció màquina (per tant atòmica) de manipulació d'un boleà, si es consulta la primera vegada canvia internament de valor (test and set) retornant el valor inicial; per consultes posteriors retorna el valor actual

sense modificar-lo. Amb una instrucció màquina diferent es pot tornar a resetejar el boleà.

Sigui la classe `TestAndSet` de JAVA que emula l'instrucció de L.M. TestAndSet definida com:

```
public class TestAndSet {

    private boolean flag;

    public TestAndSet() {flag = false;}

    ... // mètodes test_and_set() i reset()

    }
```

Realitzar una implementació del mètode `test_and_set()`

9. Es vol crear un monitor que permeti intercanviar valors a parelles de processos. El monitor només té una operació: `intercanvi(int valor)`. Després de que dos processos hagin invocat `intercanvi(int valor)`, el monitor intercanvia els valors dels arguments i els retorna als processos. Suposeu que a la cua de monitor mai hi han més de 2 processos esperant.

   Implementeu el monitor.

10. Es vol dissenyar un programa que "justifiqui" text, provinent d'un procés `P1` que genera objectes de tipus `String` de longitud variable. El funcionament del programa és el següent: el procés `P1` genera un objecte de tipus `String` i l'introdueix en un monitor de tipus `Buffer_String_a_caracter` de capacitat més gran que la longitud de qualsevol objecte `String` generat. `P1` introdueix l'objecte `String` de cop, però el `Buffer_String_a_caracter` emmagatzema internament els caràcters per separat. El procés `P2` extreu els caràcters un a un, i els introdueix en un altre monitor de tipus `Buffer_caracter_a_linia`. El procés `P3` és el que extreu objectes del monitor `Buffer_caracter_a_linia`. Els objectes que el procés procés `P3` extreu són de tipus `String` i tenen longitud igual a la capacitat del monitor `Buffer_caracter_a_linia`.

    Es proposar la realització de:

    (a) El codi del mètode `put` de la classe `Buffer_String_a_caracter`.

    (b) El codi del mètode `get` de la classe `Buffer_String_a_caracter`.

    (c) El codi del mètode `put` de la classe `Buffer_caracter_a_linia`.

    (d) El codi del mètode `get` de la classe `Buffer_caracter_a_linia`.

11. Un compte d'estalvis és compartit entre diferents processos. Cada procés pot treure o dipositar diners. El balanç del compte mai pot ser negatiu. Construir un monitor Signal and Continue que resolgui aquest problema amb les operacions `dipositar(quantitat)` i `extreure(quantitat)`.

    Es proposa la implementació de:

(a) El codi de l'operació `dipositar`.

(b) El codi de l'operació `extreure`.

12. Si es modifica el monitor del problema 11 per garantir que les operacions son resoltes per ordre d'arribada, és a dir, si hi ha 1000 i arriba un procés que vol treure 2000 i a continuació arriba un altre procés que vol treure 500, aquest últim procés s'haurà d'esperar a que el que en vol 2000 les pugui treure.

    S'hauria de realitzar

    (a) El codi de l'operació `dipositar`.

    (b) El codi de l'operació `extreure`.

13. Suposem que el següent monitor s'utilitza per controlar l'accés de varis processos a un recurs compartit.

```
class ControlAcces extends Monitor {
    protected boolean lliure = true;
    protected Condition torn;

    // Aquirir recurs
    public void adquirir(){
        mon_enter();
        if (!lliure) {
            torn.wait();
        }
        lliure = false;
        mon_exit();
    }

    // Alliberar recurs
    public void alliberar(){
        mon_enter();
        lliure = true;
        torn.signal();
        mon_exit();
    }
}
```

    Amb quina disciplina funciona correctament aquest monitor?

14. Es vol simular un mecanisme de pas de missatges assíncron en JAVA. A tal efecte es construeix la següent classe `Canal`. El mètode `rebre` és bloquejant en el cas en que no existeixin missatges en el canal.

```
public class Canal {
    private int num_miss;
    private Vector missatges;
```

```
    public Canal(){
        num_miss = 0;
        missatges = new Vector();
    }

    ... enviar (Object o) {
        ...
    }

    ... rebre() {
        ...
    }
}
```

Es demana implementar el codi dels mètodes `enviar` i `rebre`, de manera que les respostes siguin compatibles entre si.

15. **ProductorsABCs:** Una aplicació disposa de tres tipus de processos. Els processos `A` produeixen `a`'s, els processos `B` produeixen `b`'s i els processos `C` necessiten (consumeixen) una `a` i una `b` per a produir `ab`'s. Es vol resoldre el problema ProductorsABCs, sense exclusió mútua i amb possibilitat de més d'una producció pendent, utilitzant monitors. El codi dels processos és:

```
Procés A:
    //produir  a
    monitor.pa();

Procés B:
    //produir  a
    monitor.pb();

Procés C:
    monitor.pab();
    //produir  ab
```

Implementar el codi del monitor.

16. En un sistema hi ha dos tipus de processos concurrents `NoPrior` i `Prior` que accedeixen en exclusió mútua a un recurs comú. Els processos `Prior` tenen prioritat sobre els NoPrior, és a dir, quan arriba un procés `Prior` passa davant de tots els `NoPrior` que puguin estar esperant per accedir al recurs.

    Es vol resoldre aquest sistema fent servir monitors de Java. D'aquesta forma es defineix una classe amb els següents atributs i les seves inicialitzacions que gestiona l'accés a través dels mètodes `demanar_recurs()` i `alliberar_recurs()`:

```
//Número de processos Prior en espera:
int cuaP=0;
```

```
//Número de processos NoPrior en espera:
int cuaNoP=0;
//Indica si algun procés està accedint al recurs:
boolean ocupat=false;
```

(a) Implementar el mètode `alliberar_recurs`.

(b) Si el paràmetre `tipus` representa el tipus de procés (0=`NoPrior` i 1=`Prior`), implementar el mètode `demanar_recurs(int tipus)`.

Part 3

Pas de missatges

1. Un thread de classe `B` vol veure incrementat un comptador en `N` unitats. Per això s'ajuda de `N` threads de classe `A` que s'executen concurrentment i incrementen cadascun d'ells el comptador en una unitat. Aquest comptador pren el valor inicial zero. Els threads no comparteixen memòria i es comuniquen a través d'un parell de sockets connectats. Els threads `A` comparteixen un socket d'un extrem anomenat `socket_A` i el thread `B` usa el socket de l'altre extrem anomenat `socket_B`.

   (a) Implementar el codi dels threads `A`

   (b) Implementar el codi dels threads `B`

2. `N` threads de classe `A` s'executen concurrentment amb un thread de classe `B`. Es vol que el thread `B` s'executi quan els `N` threads `A` hagin acabat. Per això es disposa d'un parell de sockets connectats on un d'ells, `socket_A`, és compartit pels threads `A`, el thread `B` fa servir `socket_B`.

   Es demana

   (a) La codificació del mètode `run()` de la classe `A`.

   (b) La codificació del mètode `run()` de la classe `B`.

3. Es vol programar el problema d'exclusió mútua distribuït amb control d'accés centralitzat en un àrbitre (monitor actiu) concurrent. `out` és de tipus `PrintWriter` i `in` de tipus `BufferedReader`.

   Es proposa realitzar

   (a) El protocol de comunicació del Client.

   (b) El fragment de codi dels threads de l'àrbitre que dialoguen amb el Client.

4. Symetric Socket: Es vol fer servir un model simètric de connexions punt a punt amb un thread a cada extrem. Un thread defineix un socket amb la funció `Socket connect(String conn_name)` que retorna un socket de la connexió de nom `conn_name` connectat amb el socket de l'altre extrem. Fins que no es connecta, la funció bloqueja el thread. Lògicament, un

extrem haura de ser el socket actiu (`Socket`) i l'altre s'obtindrà a partir del socket passiu (`ServerSocket`).

Per a la implementació del model es fa servir un thread servidor amb nom `SERVERNAME` i port `SERVERPORT` públics que manté un conjunt de parells `(conn_name, (peername, peerport))` formats per el nom de la connexió i el nom i port del host de l'altre extrem (peer). Aquest conjunt és una taula de hash.

Si l'extrem és el socket passiu, solicita al sistema la obtenció d'un port no usat. Això es fa passant 0 al constructor `ServerSocket` i obtenint el port assignat amb `getLocalPort`. També obté el nom de host amb `getHostName`.

Es demana:

(a) La programació de `connect`.
`in` i `out` són els streams d'Input i Output de `sock`. `str` és un `String` local de `connect`.

(b) La programació del Servidor que dialoga amb `connect`.
`in` i `out` són els streams d'Input i Output de `sock`. `ht` és la taula de hash. `Peer` és la classe formada per el parell `(peername, peerport)`.

5. Un monitor actiu utilitza el mecanisme de pas de missatges per emular el funcionament dels monitors convencionals de memòria compartida. En aquest cas el monitor es converteix en un procés més de forma que l'entrada a monitor, sortida de monitor i l'equivalent al `wait()` i `notify()` són el resultat de l'intercanvi de missatges entre la resta de processos i el procés monitor.

   Partint del supòsit que contem amb els mètodes `send(dada)` i `dada = receive()` a on el mètode `send()` no és bloquejant, però el mètode `receive()` sí que ho és:

   (a) Programar la part del client i la del monitor actiu de l'emulació d'entrada a monitor amb un monitor actiu monothread.

   (b) Programar la part del client i la del monitor actiu del `wait()` amb un monitor actiu monothread.

6. Es vol realitzar un servidor que faci les funcions d'un buffer de missatges amb capacitat ilimitada, al qual els clients interaccionen amb pas de missatges. El codi principal del servidor és el següent:

```
public class Server {

    static Semaphore mutex;
    static Shared shared;

    public static void main(String[] args) {
        int port = Integer.parseInt(args[0]);
        mutex = new Semaphore(1);
        shared = new Shared();
```

```
        try {
            ServerSocket srv_sock = new ServerSocket(port);
            while (true) {
                Socket conn_sock = srv_sock.accept();
                new Service(conn_sock).start();
            }
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}

class Service extends Thread {

    BufferedReader input;
    PrintWriter output;

    Service(Socket sock) throws IOException {
        input = new BufferedReader(
                    new InputStreamReader(
                                sock.getInputStream()
                    )
        );
        output=new PrintWriter(sock.getOutputStream(),true);
    }

    public void run() {
        try {
            while (true) {
                String msg = input.readLine();
                if (msg == null) break;
                if (msg.equals("put")) {
                    msg = input.readLine();
                    Server.mutex.P();
                    Server.shared.process_put(this,msg);
                    Server.mutex.V();
                } else if (msg.equals("get")) {
                    Server.mutex.P();
                    Server.shared.process_get(this);
                    Server.mutex.V();
                } else if (msg.equals("size")) {
                    Server.mutex.P();
                    Server.shared.process_size(this);
                    Server.mutex.V();
                }
            }
        } catch (IOException e) {
            System.out.println(e);
        }
    }
```

```
}
```

Es proposa realitzar:

(a) El codi de la classe `Shared`.

(b) El fragment de codi corresponent a l'operació de posar un missatge en el buffer. Respecte al codi dels possibles clients, suposarem que `input` i `output` són el `BufferedReader` i el `PrintWriter` corresponents al socket connectat amb el servidor.

(c) El fragment de codi corresponent a l'operació de treure un missatge del buffer

7. **One lane bridge:** Per un pont poden circular cotxes, modelats com a threads, en un o altre sentit, però no en ambdos sentits a la vegada. Es programa una solució amb pas de missatges pura amb clients remots, proxies o agents en l'extrem servidor i monitor actiu que fa el control d'accés. Els clients realitzen l'operació `enter(id, turn)` per solicitar l'entrada del cotxe `id` al pont en sentit `turn` i `exit()` per sortir del pont. El codi del monitor actiu usa una variable `l` de tipus `ArrayList`, operacions `read_str`, `read_int` i `write_char` per llegir un `String`, un `int` i escriure un `char` a través d'un `Socket` respectivament i respon al següent esquema:

```
while (true) {
    if (cond) {
        A
    } else {
        B
    }
    if (str.equals("enter")) {
        C
    } else {
        D
    }
}
```

Us proposa d'implementar:

(a) El codi `if (cond) A`.

(b) El codi `B`.

(c) El codi `C`.

(d) El codi `D`.

8. En un entorn distribuït hi ha `N` processos que es comuniquen per pas de missatges. Cada procés te un identificador diferent, `ID` (amb valors de `0` a `N-1`), i un número enter `meu_val` (amb un valor positiu qualsevol). Es vol construir una aplicació per aconseguir que tots els processos puguin conèixer el valor (`meu_val`) més gran. Per fer-ho es considera un anell (virtual), on el procés `ID` rep el màxim provisional, calculat per el procés `ID-1`, compara aquest valor amb el seu valor, calcula el màxim dels dos

valors i l'envia al procés següent, `ID+1 mod N`. Després d'una volta, el primer procés (`ID=0`) ja rep el màxim. Finalment es fa una segona volta on aquest màxim es passa a tots els processos.

L'esquelet del codi de cadascun dels processos és el següent:

```
public class proces extends Thread{
    int ID; //Identificador del procés
    int meu_val; //Valor emmagatzemat per el procés
    int val_max; //Valor màxim provisional
    int valor_maxim; //Valor màxim de tots els processos
    int N; //Número de processos

    //El procés rep l'identificador i el seu valor.
    public proces(int ID, int meu_val, int N){
        this.ID=ID;
        this.meu_val=meu_val;
        this.N=N;
    }

    private int llegir_enter(Socket s){
       ....
       //Retorna un enter llegit de s
    }

    private void escriure_enter(Socket s, int en){
       ....
       //escriu un enter a s
    }

    private int esperar_valor(){
       ...
    }

    private int comparar_valor( int val){
       ...
    }

    private void enviar_valor(int val_max){
       ...
    }

    private void enviar(int val_max){
       ...
    }

    public void run(){
        int val;

        //primera volta
        val_max=meu_val;
```

```
        val=esperar_valor();
        val_max=comparar_valor(val);
        enviar_valor(val_max);

        //segona_volta
        valor_maxim=esperar_valor();
        enviar_valor(valor_maxim);
    }
}
```

Es demana la realització de:

(a) El codi del mètode `esperar_valor()`.

(b) El codi del mètode `comparar_valor(int val)`.

(c) El codi del mètode `enviar_valor(int val_max)`.

(d) El codi del mètode `enviar(int val)`.

Part 4

Aspectes de concurrència

1. Siguin les classes definides com:

```
class Fil extends Thread {
    String missatge;

    Fil(String m){
        missatge = m;
    }

    public void run(){
        while(true){
            System.out.println(missatge);
        }
    }
}

public class ExecutaFils {
    public static void main(String args[]){
        Fil filA = new Fil("A");
        Fil filB = new Fil("B");
        filA.run();
        filB.start();
    }
}
```

   Quina és la sortida del programa?