Programación concurrente

Práctica 1 - Variables compartidas

1)

a) Verdadera. El valor de x al terminar el programa es 56 en el siguiente caso:
   - Se ejecuta P1 completo (x vale 10 al finalizar)
   - Se ejecuta P2 completo (x vale 11 al finalizar)
   - Se ejecuta P3 completo (x vale 56 al finalizar)

b) Verdadero. El valor de x al terminar el programa es 22 en el siguiente caso:
   - Se ejecuta P1, excepto la asignación en x del valor de y + 2 (x vale 0)
   - Se comienza a ejecutar P3. Se obtiene el valor de x (que es 0) para multiplicarlo por 3
   - Se termina de ejecutar P1 (x vale 10)
   - Se termina de ejecutar P#. Se obtiene el valor de x (que es 10) para multiplicarlo por 2 y se le suma 1 (x vale 21 (x:=0*3+10*2+1))
   - Se ejecuta P2 completo (x vale 22 al finalizar)

c) Verdadero. El valor de x al terminar el programa es 23 en el siguiente caso:
   - Se ejecuta P1, excepto la asignación en x del valor de y + 2 (x vale 0)
   - Se comienza a ejecutar P3. Se obtiene el valor de x (que es 0) para multiplicarlo por 3
   - Se termina de ejecutar P1 (x vale 10)
   - Se ejecuta P2 completo (x vale 11 al finalizar)
   - Se termina de ejecutar P3. Se obtiene el valor de x (que es 11) para multiplicarlo por 2 y se le suma 1 (x vale 23 (x:=0*3+11*2+1))

2)

```
cantidad_veces = 0;
int N; # Número a verificar
int M; # M>=1
int arreglo[1..M]; # Arreglo de longitud M

Process VerificarApariciones [id=1..M] {
     if(arreglo[id]==N){
          <cantidad_veces = cantidad_veces + 1>
```

```
      }
}
```

3)

a) No es correcta la solución. Con las siguientes modificaciones si funcionará:

```
int cant = 0;
int pri_ocupada = 0;
int pri_vacia = 0;
int buffer[N];

Process Productor{
      while (true){
            producirElemento()
            <await (cant < N);>
            buffer[pri_vacia] = elemento;
            <cant++;>;
            pri_vacia = (pri_vacia + 1) mod N;
      }
}
Process Consumidor::{
      while (true){
            <await (cant > 0);>
            elemento = buffer[pri_ocupada];
            <cant --;>;
            pri_ocupada = (pri_ocupada + 1) mod N;
            consumirElemento();
      }
}
```

b)

```
int cant = 0;
int pri_ocupada = 0;
int pri_vacia = 0;
int buffer[N];

Process Productor [1..P]{
      while (true){
            producirElemento()
            <await (cant < N); {
                  buffer[pri_vacia] = elemento;
                  cant++;
                  pri_vacia = (pri_vacia + 1) mod N;
            }>;
      }
}
Process Consumidor [1..C]{
      while (true){
            <await (cant > 0); {
                  elemento = buffer[pri_ocupada];
                  cant --;
                  pri_ocupada = (pri_ocupada + 1) mod N;
            }>;
            consumirElemento();
      }
}
```

4)

```
Recurso cola[1..5]; # Cola con 5 instancias de un recurso
```

```
Process Recurso [id = 1..N]{
      while(true){
            if(necesitaUsarRecurso()){
                  <await !cola.empty(); {
                        Recurso recurso = cola.dequeue()
                  }>;
                  usarRecurso()
                  <cola.enqueue(recurso);>;
            }
      }
}
```

5)

a)

```
bool impresora_libre = true

Process Persona [id = 1..N] {
      <await (impresora_libre); impresora_libre = false;>
      Imprimir(documento);
      impresora_libre = true;
}
```

b)

```
int cola[N];

Process Persona [id = 1..N] {
      <cola.enqueue(id);>;
      <await cola.peek() == id;>;
```

```
      Imprimir(documento);
      <cola.dequeue();>;
}
```

c)

```
int persona_actual = 1;

Process Persona [id = 1..N] {
      <await persona_actual == id;>
      Imprimir(documento);
      persona_actual = persona_actual + 1;
}
```

d)

```
int cola[N];
int es_turno_de = -1;

Process Persona [id = 1..N] {
      <cola.enqueue(id);>;
      <await es_turno_de == id;>
      Imprimir(documento);
      es_turno_de = -1;
}

Process Coordinador {
      while (true){
            <await (!cola.empty()); es_turno_de = cola.dequeue();>;
            <await es_turno_de == -1;>;
      }
}
```

6)

Propiedades a cumplir:

- Exclusión mutua: se cumple ya que, el proceso SC1 solo puede estar en su SC cuando la variable turno es 1, pero para que la variable turno sea 1, el proceso SC2 no puede estar en su SC (ya que hace turno = 1 al salir de la misma y no podrá entrar nuevamente hasta que el proceso SC1 salga de su SC y haga turno = 2)
- Ausencia de deadlock (livelock): se cumple ya que para que haya deadlock, ambos procesos deberían bloquearse en su protocolo de entrada (while…), pero para que eso suceda, turno debería ser 1 y 2 a la vez.
- Ausencia de demora innecesaria: se cumple, ya que si uno de los procesos está fuera de su SC (nunca terminan), significa que la variable turno fue modificada para que el otro proceso acceda a su SC.
- Eventual entrada: se cumple ya que, al ser SC un segmento de código finito, el cambio de variable para permitir que el otro proceso entre a su SC ocurrirá y, eventualmente, el otro proceso entrará a su SC

7)

```
bool quiere_entrar[1..N] = ([N] false)
int actual = -1;

Process SC [id = 1..N] {
     while (true){
          if(necesitaEntrarASC(id)){ # El proceso quiere entrar a
su SC
               quiere_entrar[id] = true;
               while (actual != id){ # Esperando a que lo dejen
entrar
                    skip;
               }
               # Lo dejan entrar
```

```
                SC;
                actual = -1; # Se saca el acceso a la SC
                quiere_entrar[id] = false; # Avisa que terminó
           }
     }
}

Process Coordinador {
     while (true){
          for (i = 1; i<=N; i++){
                if(quiere_entrar[i]){  #  El  proceso  SC[i]  quiere
entrar
                     actual = i; # Lo dejo entrar
                     while  (quiere_entrar[i]){ # Esperando a que
el proceso SC[i] termine su SC
                          skip;
                     }
                     # El proceso SC[i] terminó su SC
                }
           }
     }
}
```