Ejercicio 1

Para el siguiente programa concurrente, suponga que todas las variables están inicializadas en 0 antes de empezar. Indique cuál/es de las siguientes opciones son verdaderas:

a. En algún caso el valor de x al terminar el programa es 56.
b. En algún caso el valor de x al terminar el programa es 22.
c. En algún caso el valor de x al terminar el programa es 23.

P1::
`If (x = 0) then`
`  y:= 4*2;`
`  x:= y + 2;`
	P2::
	`If (x > 0) then`
	`  x:= x + 1;`
		P3::
		`x := (x * 3) +`
		`  (x * 2) + 1;`

1. acc = x
2. si acc == 0:
  a. y = 8
  b. x = 10
	3. acc = x
	4. si acc > 0:
	  a. x = x + 1
		5. acc1 = x * 3
		6. acc2 = x * 2
		7. acc1 + acc2 + 1

Historias posibles:

1. **x = 56** 1 → 2 → 3 → 4 → 5 → 6 → 7
2. **x = 22** 1 → 5 → 2 → 6 → 7 → 3 → 4
3. **x = 23** 1 → 5 → 2 → 3 → 4 → 6 → 7

Ejercicio 2

Realice una solución concurrente de grano grueso (utilizando <> y/o <await B; S>) para el siguiente problema. Dado un número N, verifique cuantas veces aparece ese número en un arreglo de longitud M.

```
// N: número a buscar. M: longitud del arreglo. P: cantidad de procesos.
// Asumiendo que M es múltiplo de P.

total := 0;

process buscador[id: 0 .. P-1] {
    tamañoSegmento := M / P
    inicio := tamañoSegmento * id;
    coincidencias := 0;
    for i := inicio to (inicio + tamañoSegmento - 1) {
        if (arreglo[i] == n) {
            coincidencias++;
        }
    }
    if (coincidencias > 0) <total += coincidencias>;
}
```

Ejercicio 3

Dada la siguiente solución de grano grueso:

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

process productor {
  while (true) {
    produce elemento
    <await (cant < N); cant++>
    buffer[pri_vacia] = elemento;
    pri_vacia = (pri_vacia + 1) mod N;
  }
}

Process consumidor {
  while (true) {
    <await (cant > 0); cant-- >
    elemento = buffer[pri_ocupada];
    pri_ocupada = (pri_ocupada + 1) mod N;
    consume elemento
  }
}
```

Inciso a

Indicar si el siguiente código funciona para resolver el problema de Productor/Consumidor con un buffer de tamaño N. En caso de no funcionar, debe hacer las modificaciones necesarias.

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

process productor {
  while (true) {
    produce elemento
    <await (cant < N);
      cant++;
      buffer[pri_vacia] = elemento;
    >
    pri_vacia = (pri_vacia + 1) mod N;
  }
}

Process consumidor {
  while (true) {
    <await (cant > 0);
      cant--;
```

```
      elemento = buffer[pri_ocupada];
    >
    pri_ocupada = (pri_ocupada + 1) mod N;
    consume elemento
  }
}
```

Inciso b

Modificar el código para que funcione para C consumidores y P productores.

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

process Productor[idP: 1..P] {
  while (true) {
    produce elemento;
    <await (cant < N);
      cant++;
      buffer[pri_vacia] = elemento;
      pri_vacia = (pri_vacia + 1) mod N
    >;
  }
}

process Consumidor[idC: 1..C] {
  while (true) {
    <await (cant > 0);
      cant--;
      elemento = buffer[pri_ocupada];
      pri_ocupada = (pri_ocupada + 1) mod N;
    >;
    consume elemento;
  }
}
```

Ejercicio 4

Realice una solución concurrente de grano grueso (utilizando <> y/o <await B; S>) para el siguiente problema. Un sistema operativo mantiene 5 instancias de un recurso almacenadas en una cola, cuando un proceso necesita usar una instancia del recurso la saca de la cola, la usa y cuando termina de usarla la vuelve a depositar.

```
const int CANT_INSTANCIAS = 5;
Recurso cola[CANT_INSTANCIAS] = {...};

process proceso[id: 0..N] {
  Recurso recurso;
  <await !cola.isEmpty(); recurso = cola.pop()>
  usar recurso
  <cola.push(recurso)>
}
```

Ejercicio 5

En cada ítem debe realizar una solución concurrente de grano grueso (utilizando <> y/o <await B; S>) para el siguiente problema, teniendo en cuenta las condiciones indicadas en el item. Existen N personas que deben imprimir un trabajo cada una.

Inciso a

Implemente una solución suponiendo que existe una única impresora compartida por todas las personas, y las mismas la deben usar de a una persona a la vez, sin importar el orden. Existe una función Imprimir(documento) llamada por la persona que simula el uso de la impresora. Sólo se deben usar los procesos que representan a las Personas.

```
Impresora impresora = ...;
bool impresoraDisponible = true;

process proceso[id: 1..N] {
  Documento documento = ...;
  <await impresoraDisponible; impresoraDisponible = false>
  impresora.imprimir(documento)>;
  impresoraDisponible = true;
}
```

Inciso b

Modifique la solución de (a) para el caso en que se deba respetar el orden de llegada.

```
Impresora impresora = ...;
int turnoActual = 0, utlimoTurno = 0;

process proceso[id: 1..N] {
  Documento documento = ...;
  int miTurno;
```

```
  <miTurno = ultimoTurno++>;

  <await turnoActual == miTurno;>
  impresora.imprimir(documento);
  turnoActual += 1;
}
```

Inciso c

Modifique la solución de (a) para el caso en que se deba respetar el orden dado por el identificador del proceso (cuando está libre la impresora, de los procesos que han solicitado su uso la debe usar el que tenga menor identificador).

```
Impresora impresora = ...;
Cola esperandoTurno = ...;
int turnoActual = 0;

process proceso[id: 1..N] {
  Documento documento = ...;

  <if(turnoActual == 0) {
    turnoActual = id;
  } else {
    esperandoTurno.insertarOrdenado(id);
  }>

  <await turnoActual == id>
  imprimir(impresora, documento);

  <if cola.empty() {
     turnoActual= 0;
  } else {
     turnoActual=esperandoTurno.pop();
  }>
}
```

```
Impresora impresora = ...;
Cola esperandoTurno = ...;

process proceso[id: 1..N] {
  Documento documento = ...;
  <esperandoTurno.insertarOrdenado(id)>;
  <await esperandoTurno.first() == id>;
  impresora.imprimir(documento);
  <esperandoTurno.eliminar(id);>
```

```
}
```

Inciso d

Modifique la solución de (a) para el caso en que se deba respetar estrictamente el orden dado por el identificador del proceso (la persona X no puede usar la impresora hasta que no haya terminado de usarla la persona X-1).

```
Impresora impresora = ...;
int turnoActual = 1;

process proceso[id: 1..N] {
  Documento documento = ...;

  <await id == turnoActual>;
  impresora.imprimir(documento);
  turnoActual += 1;
}
```

Inciso e

Modifique la solución de (c) para el caso en que además hay un proceso Coordinador que le indica a cada persona cuando puede usar la impresora.

```
Impresora impresora = ...;
Cola esperandoTurno = ...;
int turnoActual = 0;
bool impresoraDisponible = true;

process coordinador {
  for 0 to N {
    <await (!cola.isEmpty(); turnoActual = esperandoTurno.pop()>
    <await impresoraDisponible>
    impresoraDisponible = false;
  }
}

proces persona[id: 1..N] {
  Documento documento = ...;
  <cola.push(id)>
  <await id == turnoActual>
  imprimir(impresora, documento);
  impresoraDisponible = true;
}
```

Ejercicio 6

Resolver con SENTENCIAS AWAIT (<> y/o <await B; S>) el siguiente problema. En un examen final hay P alumnos y 3 profesores. Cuando todos los alumnos han llegado comienza el examen. Cada alumno resuelve su examen, lo entrega y espera a que alguno de los profesores lo corrija y le indique la nota. Los profesores corrigen los exámenes respectando el orden en que los alumnos van entregando.

```
const int CANT_PROFESORES = 3;
const int CANT_ALUMNOS = P;

Cola<Record{Examen, int}> examenesResuletos = new Cola();
int notas[1..CANT_ALUMNOS] = ([n] = -1);
int cantPresentes = 0;

process alumno[id: 1..CANT_ALUMNOS] {
  Examen examen = new Examen();

  <cantPresentes += 1>;
  <await (cantPresentes == CANT_ALUMNOS)>;

  resolver(examen);
  <examenesResueltos.push({ examen, id })>;

  <await (notas[id] != -1)>;
  verNota(notas[id]);
}

process profesor[id: 1..CANT_PROFESORES] {
  while (true) {
    <await (!examenesResueltos.isEmpty())
      examenResuelto = examenesResueltos.pop()
    >
    notas[examen.id] = corregir(examenResuleto.examen)
  }
}
```

Ejercicio 7

Dada la siguiente solución para el Problema de la Sección Crítica entre dos procesos (suponiendo que tanto SC como SNC son segmentos de código finitos, es decir que terminan en algún momento), indicar si cumple con las 4 condiciones requeridas:

```
int turno = 1;
```

```
process SC1 {
  while (true) {
    while (turno == 2) skip;
    SC;
    turno = 2;
    SNC;
  }
}
```

```
process SC2 {
  while (true) {
    while (turno == 1) skip;
    SC;
    turno = 1;
    SNC;
  }
}
```

1. **Exclusión mutua:** a lo sumo un proceso está en su SC.
2. **Ausencia de deadlock:** si dos o más procesos tratan de entrar a sus SC, y esas SC está libre, al menos uno tendrá exito.
3. **Ausencia de demora innecesaria:** si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC.
4. **Eventual entrada:** un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará).

Como la variable turno está inicializada con el valor 1, siempre SC1 será el primer proceso en llegar a su sección crítica, ya que nunca se cumplirá la condición del while. Por otro lado, SC2 siempre va a tener que esperar a que SC1 termine su sección crítica par poder salir de la iteración y comenzar a ejecutar su sección crítica.

Ejercicio 8

Desarrolle una solución de grano fino usando solo variables compartidas *(no se puede usar las sentencias await ni funciones especiales como TS o FA)*. Con base en lo visto en la clase 2 de teoría, resuelva el problema de acceso a sección crítica usando un proceso coordinador.

En este caso, cuando un proceso SC[i] quiere entrar a su sección crítica, le avisa al coordinador, y espera a que éste le dé permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador. Nota: puede basarse en la solución para implementar barreras con Flags y Coordinador vista en la teoría 3.

```
const int CANT_PROCESOS = N

bool esperando[0 .. CANT_PROCESOS - 1] = ([n] false);
bool continuar[0 .. CANT_PROCESOS - 1] = ([n] false);

process SC[id = 0 .. CANT_PROCESOS - 1] {
    while (true) {
        // SNC
        esperando[id] = true;
        while(!continuar[id]) skip;
        // SC
```

```
        continuar[id] = false;
        // SNC
    }
}

process coordinador {
    int id = 0;
    while (true) {
        while (!esperando[id]) id = (id + 1) % CANT_PROCESOS;
        esperando[id] = false;
        while (continuar[id]) skip;
        id = (id + 1) % CANT_PROCESOS;
}
```