## CONSIDERACIONES PARA RESOLVER LOS EJERCICIOS DE PMA

* Los canales son compartidos por todos los procesos.
* Cada canal es una cola de mensajes, por lo tanto el primer mensaje encolado es el primero en ser atendido.
* Por ser pasaje de mensajes **asincrónico** el **send no bloquea al emisor**.
* Se puede usar la sentencia **empty** para saber si hay algún mensaje en el canal, pero no se puede consultar por la cantidad de mensajes encolados.
* Se puede utilizar el **if/do no determinístico** donde cada opción es una condición boolena donde se puede preguntar por variables locales y/o por empty de canales.

*if (cond 1) -> Acciones 1;*

*\*(cond 2) -> Acciones 2;*

*....*

*\*(cond N) -> Acciones N;*

*end if*

De todas las opciones cuya condición sea Verdadera elige una en forma no determinística y ejecuta las acciones correspondientes. Si ninguna es verdadera sale del if/do si hacer nada.

* Se debe tratar de evitar hacer busy waiting (sólo hacerlo si no hay otra opción).
* En todos los ejercicios el tiempo debe representarse con la función delay.

## 1. Supongamos que tenemos una abuela que tiene dos tipos de lápices para dibujar: 10 de colores y 15 negros. Además tenemos tres clases de niños que quieren dibujar con los lápices: los que quieren usar sólo los lápices de colores (tipo C), los que usan sólo los lápices negros (tipo N), y los niños que usan cualquier tipo de lápiz (tipo A).

## a) Implemente un código para cada clase de niño de manera que ejecute pedido de lápiz, lo use por 10 minutos y luego lo devuelva y además el proceso abuela encargada de asignar los lápices.

chan **pedirColor**(int id);

chan **pedirNegro**(int id);

chan **pedirAleatorio**(int id);

chan **devolverColor**;

chan **devolverNegro**;

chan [1..N] niños;

### Process NiñoA [id= 1..A] {

**begin**

while true {

send **pedirAleatorio**(id);

receive niños[id](lapiz);

delay (10);

if (lapiz = “negro”) {

send **devolverNegro**();

} else {

send **devolverColor**();

}

} //while

} niñoA

### Process NiñoN [id= 1..N] {

**begin**

while true {

send **pedirNegro**(id);

receive niños[id](negro);

delay (10);

send **devolverNegro**(negro);

} //while

} //niñoN

### Process NiñoC [id= 1..C] {

**begin**

while true {

send **pedirColor**(id);

receive niños[id](color);

delay (10);

send **devolverColor**(color);

}//while

} //niñoC

### Process Abuela {

**var**

int cantNegro = 15;

int cantColor = 10;

**begin**

while true {

if (not empty(**pedirColor**) && (cantColor > 0) {

→ receive **pedirColor**(id);

cantColor --;

send niños[id](color);

}

\* (not empty(**pedirNegro**) && (cantNegro > 0) {

→ receive **pedirNegro**(id);

cantNegro --;

send niños[id](negro);

}

\* (not empty(**devolverColor**) {

→ receive **devolverColor**(color);

cantColor ++;

}

\* (not empty(**devolverNegro**) {

→ receive **devolverNegro**(negro);

cantNegro ++;

}

\* (not empty(**pedirAleatorio**) && (cantColor > 0) {

→ receive **pedirAleatorio**(id);

cantColor --;

send niños[id](color);

}

\* (not empty(**pedirAleatorio**) && (cantNegro > 0) {

→ receive **pedirAleatorio**(id);

cantNegro --;

send niños[id](negro);

}

} //while

} //abuela

## b) Modificar el ejercicio para que a los niños de tipo A se les puede asignar un lápiz sólo cuando: hay lápiz negro disponible y ningún pedido pendiente de tipo N, o si hay lápiz de color disponible y ningún pedido pendiente de tipo C. NOTA: se deben modelar los procesos niño y el proceso abuela.

Se deben modificar algunas condiciones en los IF del proceso Abuela:

**\* ( not empty(pedirAleatorio) && (cantColor > 0) && (empty(pedirColor) ) {**

→ receive **pedirAleatorio**(id);

cantColor --;

send niños[id](color);

}

**\* ( (not empty(pedirAleatorio) && (cantNegro > 0) && (empty(pedirNegro) ) {**

→ receive **pedirAleatorio**(id);

cantNegro --;

send niños[id](negro);

}

## 2. Se desea modelar el funcionamiento de un banco en el cual existen 5 cajas para realizar pagos. Existen P personas que desean pagar. Para esto cada una selecciona la caja donde hay menos personas esperando, una vez seleccionada espera a ser atendido. NOTA: maximizar la concurrencia.

chan **pedirCaja**(int id);

chan [1..C] **numeroCaja**(int caja); //los clientes reciben el n de caja

chan [1..C] **vuelto**(float vuelto); //para que los clientes reciban el vuelto que les corresponde, en caso de haber vuelto.

chan [1..5] **pagar**(int id);

### Process Caja [id = 1..5] {

**begin**

while true {

receive **pagar**[id](idP);

send **vuelto**[idP](vuelto);

send **personaSeFue**(id);

} //while

} //caja

### Process Persona [id = 1..P] {

**begin**

// llega la persona

send **pedirCaja**(id);

receive **numeroCaja**[id](caja);

// va a la caja

send **pagar**[caja](idP);

receive **vuelto**[id](vuelto);

// se va

}

### Process Administrador :: {

**var**

array of int [1..n] cajas = [0];

**begin**

while true {

if (not empty(**pedirCaja**){

→ receive **pedirCaja**(id);

caja = *calcularCajaMinima*(cajas);

send **numeroCaja**[id](caja); // le manda a la persona la caja mínima

cajas[caja] ++;

}

\* (not empty(**personaSeFue**){

→ receive **personaSeFue**(caja);

cajas[caja] --;

}

} //while

} //administrador

## 3. En un edificio existen 3 porteros y P personas. Las personas dejan reclamos en la oficina de los porteros que deben ser atendidos por cualquiera de ellos. Cada portero está continuamente trabajando, si hay algún reclamo pendiente lo atiende, si no realiza un recorrido por los pisos durante 10 minutos.

## NOTA: la persona no debe esperar a que el reclamo sea atendido, ni se le debe avisar que se resolvió.

***Explicación ayudante****: El admin no debe preguntar al portero si está libre. El portero debe avisar que está libre y que le den un reclamo, y si no hay nada, ir a recorrer los pisos 10 min.*

chan [1..3] **darSolicitud**(int número);

chan **pedirSolicitud**(int id);

chan **hacerReclamo**(string reclamo);

**Process Portero [id = 1..3]{**

**begin**

while true {

send **pedirSolicitud**(id);

receive **darSolicitud**[id](número);

if(número = -1) {

delay(10);

} else {

//atiende la solicitud

}

} //while

} //portero

### Process Persona [id = 1..P] {

**begin**

send **hacerReclamo**(reclamo);

}

### Process Administrador :{ //Se necesita porque los porteros hacen más que una tarea: atender reclamos y hacer el recorrido por los pisos.

**begin**

while true {

receive **pedirSolicitud**(id);

if (not empty(**hacerReclamo**)) {

receive **hacerReclamo**(solicitud);

} else {

solicitud = -1;

}

→ send **darSolicitud**[id](solicitud);

} //while

} //administrador

## 4.

chan **dejarPedido**(int id, string pedido); //el cliente deja un pedido

chan [1..C] **pedidos**; //arreglo donde los clientes reciben sus pedidos. El cocinero le da un plato al cliente

chan **damePedidoV**(int id); //los vendedores toman un pedido (le piden al admin)

chan [1..2] **darPedidoV**(string pedido, int id); //el admin le da un pedido a un vendedor.

chan **darPedidoC**(string pedido, int id); //los cocineros piden un pedido a los vendedores.

**Process Vendedor [id = 1..3] {**

begin

while true {

// el vendedor solicita un pedido al admin.

send **damePedidoV**(id);

// el admin le manda un pedido al vendedor.

receive **darPedidoV[id]**(pedido, idP);

if (pedido = -1) {

delay(10); //repone pack

} else {

// le manda el pedido a un cocinero

send **darPedidoC**(pedido, idP);

}

} //while

} //vendedor

### Process Cocinero [id=1..2]{

**begin**

while true {

// el vendedor le envia un pedido de un determinado cliente al cocinero

receive **darPedidoC**(pedido, id);

delay(10); //cocinando

plato = armarPedido();

send **pedidos**[id](plato);

}

}

### Process Cliente [id = 1...C]{

**begin**

*//el cliente manda un pedido*

send **dejarPedido**(id,pedido);

*//se queda esperando a recibir su pedido. Hasta que no le hagan el send se queda bloqueado en el receive.*

receive **pedidos**[id](plato);

} //cliente

### Process Administrador :: {

**begin**

while true {

//el admin recibe una solicitud de pedido del vendedor

receive **damePedidoV**(idV);

if not(empty(**dejarPedido**)) {

receive **dejarPedido**(idC, pedido);

send **darPedidoV**[idV](idC, pedido);

} else { //no hay clientes esperando

send **darPedidoV**[idV](-1, null);

} // else

} //while

} //administrador

.

**EJEMPLOS.**

* **SI EN EL EJERCICIO FUERA UN SOLO VENDEDOR, SE PODRÍA HACER:**

**Process Vendedor: {**

**begin**

**while true**

if not(empty(dejarPedido)) {

receive dejarPedido(idC, pedido);

send darPedidoC(idC, pedido); *//le da el pedido al* cocinero

} else {

//reponer botellas

delay(10);

} // else

} //while

} //Vendedor

* **EN EL EJERCICIO TENEMOS 3 VENDEDORES, LO QUE NO SE PUEDE HACER ES:**

**Process Vendedor [id=1..3] {**

**begin**

**while true**

if not(empty(dejarPedido)) {

receive dejarPedido(idC, pedido); **//se pueden quedar acá trabados**

else {

//reponer botellas

delay(10);

} // else

} //while

} //Vendedor

**(SI EN ESTE CASO FUERA UNA SOLA TAREA, SI FUNCIONARÍA)**

## 5. Se desea modelar una competencia de atletismo. Para eso existen dos tipos de procesos: C corredores y un portero. Los corredores deben esperar que se habilite la entrada a la pista, donde deben esperar que lleguen todos los corredores para comenzar. El portero es el encargado de habilitar la entrada a la pista. NOTAS: el proceso portero NO puede contabilizar nada, su única función es habilitar la entrada a la pista; NO se puede suponer ningún orden en la llegada de los corredores al punto de partida.

### chan llegada;

chan **comenzar**;

chan **pista**;

chan **portero**;

### 

### Process Corredor [id = 1..C]{ //ESTE PROCESO IMPLEMENTA UNA BARRERA

**begin**

if (ID == 1){ // si soy el primer corredor

for (1 to C - 1) {

receive **llegada**();

send **portero**();

}

send **portero**();

send **comenzar**();

} else {

send **llegada**();

}

receive **pista**();

} //Corredor

### Process Portero:: {

**begin**

receive **comenzar**();

while(not empty(**portero**)) {

receive ***portero***();

send **pista**();

}

} //Portero

## 6. Suponga que N personas llegan a la cola de un banco. Una vez que la persona se agrega en la cola no espera más de 15 minutos para su atención, si pasado ese tiempo no fue atendida se retira. Para atender a las personas existen 2 empleados que van atendiendo de a una y por orden de llegada a las personas.

### chan llegue(int id);

array [1..N] chan **iniciarTimer**;

array [1..N] chan **rtaEmpleado**(string estado);

array [1..N] chan **avisarPersona**(string estado);

chan **quieroAtender**(int id); //el empleado quiere atender a una persona

chan [1..N] **finTiempo**;

### Process Persona [id = 1..N]{

**var**

string estado;

**begin**

send **llegue**(id); // le avisa al empleado

send **iniciarTimer**[id]();

receive **avisarPersona**[id](estado);

if (estado == “timeout”) {

//me voy

} else {

//me atendieron

}

} //Persona

### Process Estado [id = 1..N]{

**var**

string estado = “esperando”;

**begin**

//un empleado con id IdE, quiere atender a una persona con id idP.

if (!empty(**quieroAtender**[id])) {

receive **quieroAtender**[id](idE);  
 if (estado = “esperando”) {

estado = “atender”;

}

send **rtaEmpleado**[idE](estado);

}

\* (!empty(**finTiempo[id]**)){

receive **finTiempo**[id]();

if (estado = ”esperando”) {

estado = “timeout”;

send **avisarPersona**[id](estado);

}

}

} //Estado

### Process Timer [id = 1..N]{

**begin**

receive **iniciarTimer**[id]();

delay(15);

send **finTiempo**[id]();

} //Timer

### Process Empleado [id = 1..2]{ //no es necesario un administrador porque el empleado hace una sola cosa, por ende, no importa que sean dos empleados.

**begin**

while true {

receive **llegue**(idP);

send **quieroAtender**[idP](id);

receive **rtaEmpleado**[id](estado);

if (estado == ‘atender’) {

//atiende a la persona

estado = “atendiendo”;

send **avisarPersona**[idP](estado);

}

} //while

} //Empleado

PMS

## 

## CONSIDERACIONES PARA RESOLVER LOS EJERCICIOS DE PMS:

* Los canales son punto a punto y no deben declararse.
* No se puede usar la sentencia **empty** para saber si hay algún mensaje en un canal.
* Tanto el **envío** como la **recepción** de mensajes es bloqueante.
* Sintaxis de las sentencias de envío y recepción:

*Envío: nombreProcesoReceptor!port (datos a enviar)*

*Recepción: nombreProcesoEmisor?port (datos a recibir)*

El port (o etiqueta) puede no ir. Se utiliza para diferenciar los tipos de mensajes

que se podrían comunicarse entre dos procesos.

* En la sentencia de comunicación de recepción se puede usar el comodín **\*** si el

origen es un proceso dentro de un arreglo de procesos. Ejemplo:

+

*Clientes[\*]?port(datos).*

* Sintaxis de la Comunicación guardada:

***Guarda****: (condición booleana); sentencia de recepción → sentencia a realizar*

Si no se especifica la condición booleana se considera **verdadera** (la condición booleana sólo puede hacer referencia a **variables locales al proceso**).

Cada guarda tiene tres posibles estados:

* **Elegible:** la condición booleana es verdadera y la sentencia de comunicación se puede resolver inmediatamente.
* **No elegible**: la condición booleana es falsa.
* **Bloqueada**: la condición booleana es verdadera y la sentencia de comunicación no se puede resolver inmediatamente.

Sólo se puede usar dentro de un **if** o un **do** guardado:

El **IF** funciona de la siguiente manera: de todas las guardas elegibles se selecciona una en forma no determinística, se realiza la sentencia de comunicación correspondiente, y luego las acciones asociadas a esa guarda.

Si todas las guardas tienen el estado de no elegibles, se sale sin hacer nada.

Si no hay ninguna guarda elegible, pero algunas están en estado bloqueado, Se queda esperando en el if hasta que alguna se vuelva elegible.

El **DO** funciona de la siguiente manera: sigue iterando de la misma manera que el if hasta que todas las guardas sean no elegibles.

## 1. En una estación de comunicaciones se cuenta con 10 radares y una unidad de procesamiento que se encarga de procesar la información enviada por los radares. Cada radar repetidamente detecta señales de radio durante 15 segundos y le envía esos datos a la unidad de procesamiento para que los analice. Los radares no deben esperar a ser atendidos para continuar trabajando.

### process Radar [r= 1..10] {

**begin**

while true {

señal = detectarSeñalDurante15segundos();

Buffer!**enviarSeñal**(señal);

} // while

} //Radar

### process Buffer {

**var**

queue señales; string señal;

**begin**

while true {

if (true); Radar[\*]?**enviarSeñal**(señal)->

señales.push(señal);

\* not empty(señales); Unidad?**libre**->

señal = señales.pop();

Unidad!**darSeñal**(señal);

} //while

} //Buffer

### process Unidad {

**begin**

while true {

Buffer!**libre**();

Buffer?**darSeñal**(señal);

// procesa la señal

delay(random);

}

} //Unidad

## 2. Supongamos que tenemos una abuela que tiene dos tipos de lápices para dibujar: *10 de colores y 15 negros*. Además tenemos tres clases de niños que quieren dibujar con los lápices: los que quieren usar sólo los lápices de colores (tipo C), los que usan sólo los lápices negros (tipo N), y los niños que usan cualquier tipo de lápiz (tipo A).

## a) Implemente un código para cada clase de niño de manera que ejecute pedido de lápiz, lo use por 10 minutos y luego lo devuelva y además el proceso abuela encargada de asignar los lápices.

### process Abuela: {

**var** cantC = 10; cantN = 15;

**begin**

while true {

**-- NIÑO C --**

if (cantC > 0); NiñoC[\*]?**pedirColor**(id)->

cantC --;

NiñoC[id]!**darColor**(“color”);

\* if (true); NiñoC[\*]?**devolverColor**(lapiz);

cantC ++;

**-- NIÑO N --**

\* (cantN > 0); NiñoN[\*]?**pedirNegro**(id)->

cantN --;

NiñoN[id]!**darNegro**(“negro”);

\* (true); NiñoN[\*]?**devolverNegro**(lapiz)->

cantN ++;

-**- NIÑO A --**

\* (cantC > 0); NiñoA[\*]?**pedirAleatorio**(id)->

cantC --;

NiñoA[id]!**darLapiz**(“color”);

\* (cantN > 0); NiñoA[\*]?**pedirAleatorio**(id)->

cantN --;

NiñoA[id]!**darLapiz**(“negro”);

\* (true); NiñoA[\*]?**devolverNegro**(lapiz)->

cantN ++;

\* (true); NiñoA[\*]?**devolverColor**(lapiz)->

cantC ++;

} //while

} // Abuela

### process NiñoA [id = 1..A]{

**begin**

while true {

Abuela!**pedirAleatorio**(id);

Abuela?**darLapiz**(lapiz);

delay(10);

// usa el lápiz

if (lapiz = “color”){

Abuela!**devolverColor**(lapiz);

} else {

Abuela!**devolverNegro**(lapiz);

}

} //while

} //NiñoA

### process NiñoC [id = 1..C]{

**begin**

while true {

Abuela!**pedirColor**(id);

Abuela?**darColor**(lapiz);

delay(10);

// usa el lápiz

Abuela!**devolverColor**(lapiz);

} //while

} //NiñoC

### process NiñoN [id = 1..N]{

**begin**

while true {

Abuela!**pedirNegro**(id);

Abuela?**darNegro**(lapiz);

delay(10);

// usa el lápiz

Abuela!**devolverNegro**(lapiz);

} //while

} //NiñoN

## b) Modificar el ejercicio para que a los niños de tipo A se les puede asignar un lápiz sólo cuando: hay lápiz negro disponible y ningún pedido pendiente de tipo N, o si hay lápiz de color disponible y ningún pedido pendiente de tipo C. NOTA: se deben modelar los procesos niño y el proceso abuela.

### process Abuela: {

**var**

cantC = 15; cantN = 10;

queue pendientesColor; queue pendientesNegro;

**begin**

while true {

**-- NIÑO C --**

if (true); NiñoC[\*]?**pedirColor**(id)->

if (cantC > 0) {

cantC --;

NiñoC[id]!**darColor**(“color”);

} else {

pendientesColor.push(id);

}

\* (true); NiñoC[\*]?**devolverColor**(lapiz)->

if (!empty(pendientesColor)) {

id = pendientesColor.pop()

NiñoC[id]!**darColor**(“color”);

} else {

cantC ++;

}

**-- NIÑO N --**

\*if (true); NiñoN[\*]?**pedirNegro**(id)->

if (cantN > 0) {

cantN --;

NiñoN[id]!**darNegro**(“negro”);

} else {

pendientesNegro.push(id);

}

\* (true); NinoN[\*]?**devolverNegro**(lapiz)->

if (!empty(pendientesNegro)) {

id = pendientesNegro.pop()

NiñoN[id]!**darNegro**(“negro”);

} else {

cantN ++;

}

**-- NIÑO A --**

\* (empty(pendientesColor) & cantC > 0); NiñoA[\*]?**pedirAleatorio**(id)->

cantC --;

NiñoA[id]!**darLapiz**(“color”);

\* (empty(pendientesNegro) & cantN > 0); NiñoA[\*]?**pedirAleatorio**(id)->

cantN --;

NiñoA[id[!**darLapiz**(“negro”);

\* (true); NiñoA[\*]?**devolverColor**(lapiz)->

cantC ++;

\* (true); NiñoA[\*]?**devolverNegro**(lapiz)->

cantN ++;

} //while

} // Abuela

## 3. Se debe modelar la atención en una panadería por parte de 3 empleados. Hay C clientes que ingresan al negocio para ser atendidos por cualquiera de los empleados, los cuales deben atenderse de acuerdo al orden de llegada.

### process Empleado [id = 1..3] {

**begin**

while true {

Buffer!**estoyLibre**(id);

Buffer?**dameCliente**(cliente);

// atiende al cliente

delay(random);

}

**}** //Empleado

### process Cliente [id = 1..C] {

**begin**

Buffer!**llegue**(id);

//espera que lo atiendan y se va

**}** //Cliente

### process Buffer {

**var**

queue clientes;

**begin**

while true {

if (true); Cliente[\*]?**llegue**(id);

clientes.push(id);

\* not empty(clientes); Empleado[\*]?**estoyLibre**(id)->

cliente = clientes.pop();

Empleado[id]!**dameCliente**(cliente);

} //while

**}** //Buffer

## 4. Existe una casa de comida rápida que es atendida por 1 empleado. Cuando una persona llega se pone en la cola y espera a lo sumo 10 minutos a que el empleado lo atienda. Pasado ese tiempo se retira sin realizar la compra.

### Process Persona [id = 1..N] {

**begin**

Cola!**llegue**(id);

Timer[id]!**iniciar**();

// espera a que la atiendan

Estado[id]?**avisarPersona**(estado);

if (estado == “timeout”) {

//me voy

} else {

//me están atendiendo

}

}//Persona

### Process Estado [id = 1..N]{

**var**

string estado= “esperando”;

**begin**

while true {

//el empleado quiere atender a una persona.

if (true); Empleado?**quieroAtender**();  
 if (estado = “esperando”)

estado = “atender”;

}

Empleado!**darEstadoPersona**(estado);

Persona[id]!**avisarPersona**(estado);

\* Timer[id]?**finTiempo**();

if (estado = “esperando”)

estado = “timeout”;

Persona[id]!**avisarPersona**(estado);

} //while

} //Estado

### Process Timer [id = 1..N] {

**begin**

Persona[id]?**iniciar**();

delay(10);

Estado[id]!**finTiempo**();

**}** //Timer

### Process Empleado: {

**begin**

while true {

Cola!**estoyLibre**();

Cola?**darCliente**(cliente);

Estado[cliente]!**quieroAtender**();

Estado[cliente]?**darEstadoPersona**(estado);

if (estado == “atender”) {

//atiende a la persona

}

}

**}** //Empleado

### Process Cola: {

**var**

queue clientes;

**begin**

while true {

if (true); Persona[\*]?**llegue**(id)->

clientes.push(id);

\* not empty(clientes); Empleado?**estoyLibre**()->

cliente = clientes.pop();

Empleado!**darCliente**(cliente);

} //while

**}** //Cola

## 5. En un laboratorio de genética veterinaria hay 3 empleados. El primero de ellos se encarga de preparar las muestras de ADN lo más rápido posible; el segundo toma cada muestra de ADN preparada y arma el set de análisis que se deben realizar con ella y espera el resultado para archivarlo y continuar trabajando; el tercer empleado se encarga de realizar el análisis y devolverle el resultado al segundo empleado.

### Process Empleado1 {

**begin**

while true {

//arma muestra de adn

muestra = armarMuestra();

Buffer!**recibirMuestra**(muestra);

} //while

} //Empleado1

### process Buffer :{

**var**

queue cola;

**begin**

while true {

if (true); Empleado1?**recibirMuestra**(muestra);=>

cola.push(muestra);

\* (not empty(cola)); Empleado2?**enviarMuestra**();=>

muestra = cola.pop();

Empleado2!**darMuestra**(muestra);

} //while

} //Buffer

### Process Empleado2 {

**var**

queue resultados;

**begin**

while true {

Buffer!**enviarMuestra**();

Buffer?**darMuestra**(muestra);

//arma el set de análisis que se deben realizar con ella

set = armarSetAnalisis(muestra);

Empleado3!**enviarSet**(set);

Empleado3?**obtenerResultado**(resultado);

//archiva el resultado

resultados.push(resultado);

} //while

}//Empleado2

### Process Empleado3 {

**var**

string resultado;

**begin**

while true {

Empleado2?**enviarSet**(set);

//realiza los análisis

resultado = realizarAnalisis(set);

Empleado2!**obtenerResultado**(resultado);

} //while

}//Empleado3