**INSTITUTO POLITÉCNICO NACIONAL**

**UNIDAD PROFESIONAL INTERDISCIPLINARIA EN INGENIERÍA Y TECNOLOGÍAS**

**AVANZADAS**

**Asignatura:** Sistemas Operativos en Tiempo Real

**Profesor: M**aza Casas Lamberto

**Grupo:** 3MV9

**Programación Concurrente**

**Alumno: T**éllez Pérez Luis José

Contenido

[RESUMEN 3](#_Toc10287007)

[Desarrollo 3](#_Toc10287008)

[Código 1 3](#_Toc10287009)

[Código 2 6](#_Toc10287010)

[Código 3 8](#_Toc10287011)

[Código 4 10](#_Toc10287012)

[Código 5 12](#_Toc10287013)

[Código 6 14](#_Toc10287014)

[Conclusiones 16](#_Toc10287015)

[Ilustración 1 Cygwin 4](#_Toc10287024)

[Ilustración 2 Ejecución de comandos en Cygwin 5](#_Toc10287025)

[Ilustración 3 Previsualización del código en Cygwin 5](file:///C:\Users\luis_jose7\Documents\SOTR\Programación%20Concurrente.docx#_Toc10287026)

[Ilustración 4 Ejecución del código en Baci 6](file:///C:\Users\luis_jose7\Documents\SOTR\Programación%20Concurrente.docx#_Toc10287027)

[Ilustración 5 Ejecucuión código 2 8](#_Toc10287028)

[Ilustración 6 Ejecución código 3 Dekker 10](#_Toc10287029)

[Ilustración 7 Algoritmo Dekker 11](#_Toc10287030)

[Ilustración 8 Ejecución del código en GNat 13](#_Toc10287031)

[Ilustración 9 Ejecución código Dekker 2 15](#_Toc10287032)

[Ilustración 10 Ejecución algoritmo Peterson 17](#_Toc10287033)

[Tabla 1 Código 1 4](#_Toc10287040)

[Tabla 2 Código 2 7](#_Toc10287041)

[Tabla 3 Código 3 10](#_Toc10287042)

[Tabla 4 Código Dekker 1 12](#_Toc10287043)

[Tabla 5 Código Dekker 2 15](#_Toc10287044)

[Tabla 6 Código algoritmo Peterson 16](#_Toc10287045)

# RESUMEN

La idea de programación concurrente siempre ha estado asociada a los sistemas operativos: Un sólo procesador de gran capacidad debía repartir su tiempo entre muchos usuarios. La programación de estos sistemas se hacía a bajo nivel (ensamblador). Posteriormente aparecerían lenguajes de alto nivel con soporte para este tipo de programación. Su utilización y potencial utilidad se apoya en: threads o hilos, java e internet.

Definición: Se habla de concurrencia cuando ocurren varios sucesos de manera contemporánea. En base a esto, la concurrencia en computación está asociada a la “ejecución” de varios procesos que coexisten temporalmente.

# Desarrollo

Para fines de esta práctica y probar los conceptos básicos de la programación concurrente se emplearán diferentes tipos de software que facilitarán la tarea de simulación de los códigos en los que se visualizará el comportamiento que presentan dos procesos simulados simultáneamente.

## Código 1

El código con el cual se trabaja la primera parte es:

|  |
| --- |
| Count.cm |
| int n = 0;  void p() {  int temp, i;  for (i = 0; i < 3; i++) {  temp = n;  n = temp + 1;  }  }  void q() {  int temp, i;  for (i = 0; i < 3; i++) {  temp = n;  n = temp + 1;  }  }  void main() {  cobegin { p(); q(); }  cout << "The value of n is " << n << "\n";  } |

Tabla 1 Código 1

Para el primer código nos ayudaremos con el compilador Baci de Java, para poder hacer uso del mismo empleamos Cygwin para ejecutarlo. Primero asignamos un directorio donde estarán los programas, así como el compilador Baci.



Ilustración 1 Cygwin

Una vez teniendo el directorio definido, con el comando ls podemos verificar la existencia de los archivos dentro de la carpeta.

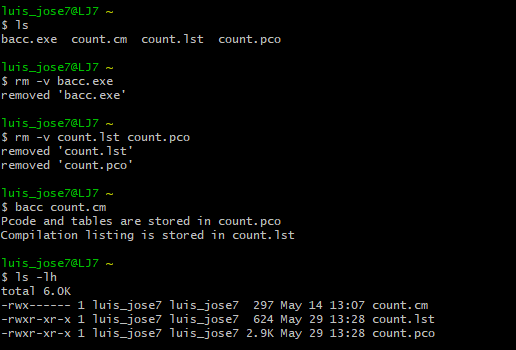


Ilustración 2 Ejecución de comandos en Cygwin

Posteriormente nos valemos de la orden cat seguido del nombre del programa a compilar. Esto abre una pre-visualización del código. Después tecleamos java –jar baci.jar en la terminal de Cygwin para ejecutar el compilador.

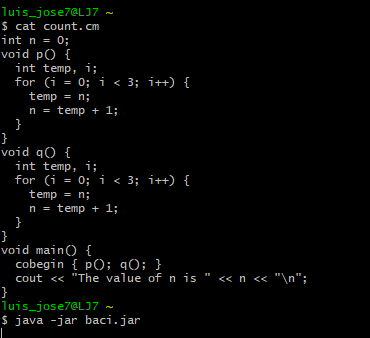


Ilustración 3 Previsualización del código en Cygwin

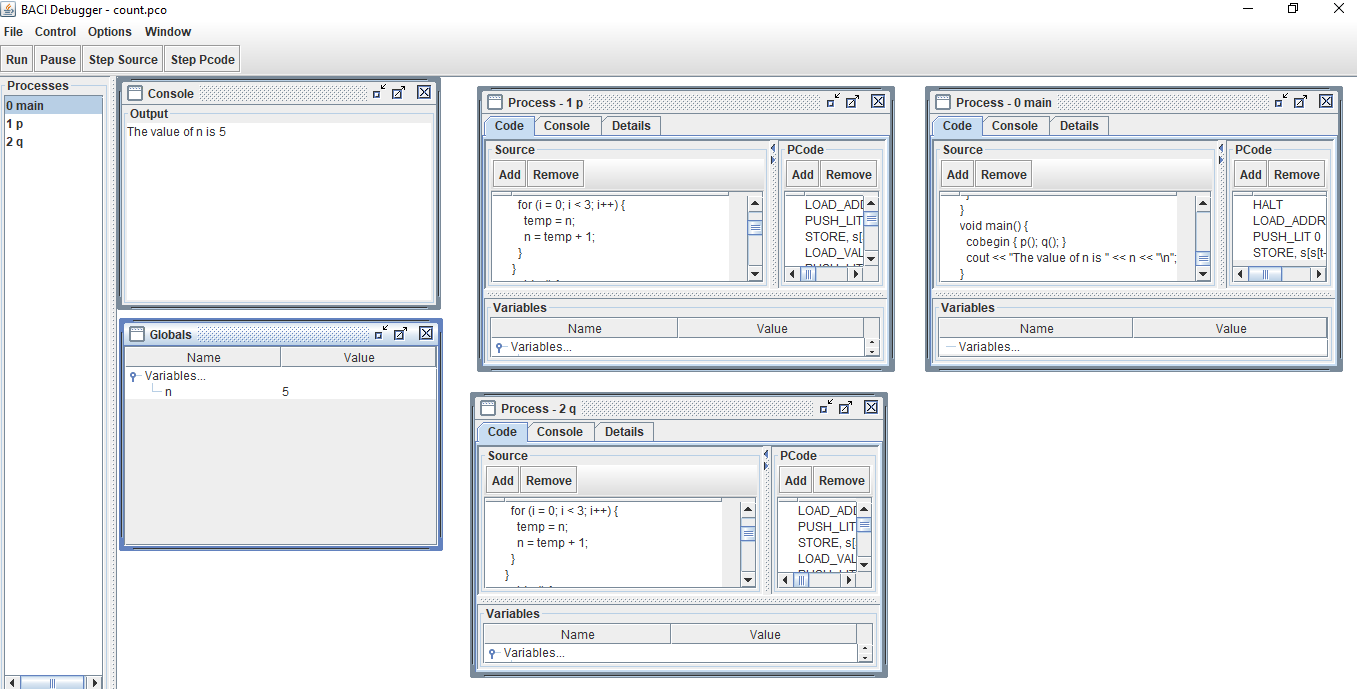
La ventana del compilador se muestra en la Ilustración, donde una vez cargado el código se corre éste para observar como los dos procesos que tiene se ejecutan concurrentemente arrojando un resultado en la ventana console.

Ilustración 4 Ejecución del código en Baci

## Código 2

El segundo código se muestra en la siguiente tabla:

|  |
| --- |
| Código 2 Count\_1 |
| |  | | --- | | int n = 0; | |  | semaphore mutex; | |  | void p() { | |  | int temp, i; | |  | for (i = 0; i < 5; i++) { | |  | wait(mutex); | |  | temp = n; | |  | n = temp + 1 | |  | signal(mutex); | |  | } | |  | } | |  | void q() { | |  | int temp, i; | |  | for (i = 0; i < 5; i++) { | |  | wait(mutex); | |  | temp = n; | |  | n = temp + 1; | |  | signal(mutex); | |  | } | |  | } | |  | void main() { | |  | initialsem(mutex,1); | |  | cobegin { p(); q(); } | |  | cout << "The value of n is " << n << | |

Tabla 2 Código 2

La ventana del compilador se muestra en la Ilustración 5, donde una vez cargado el código se corre éste para observar como los dos procesos que tiene se ejecutan concurrentemente arrojando un resultado en la ventana console. Este código hace uso de un semáforo para indicar cuando se finaliza un proceso para que inicie otro.

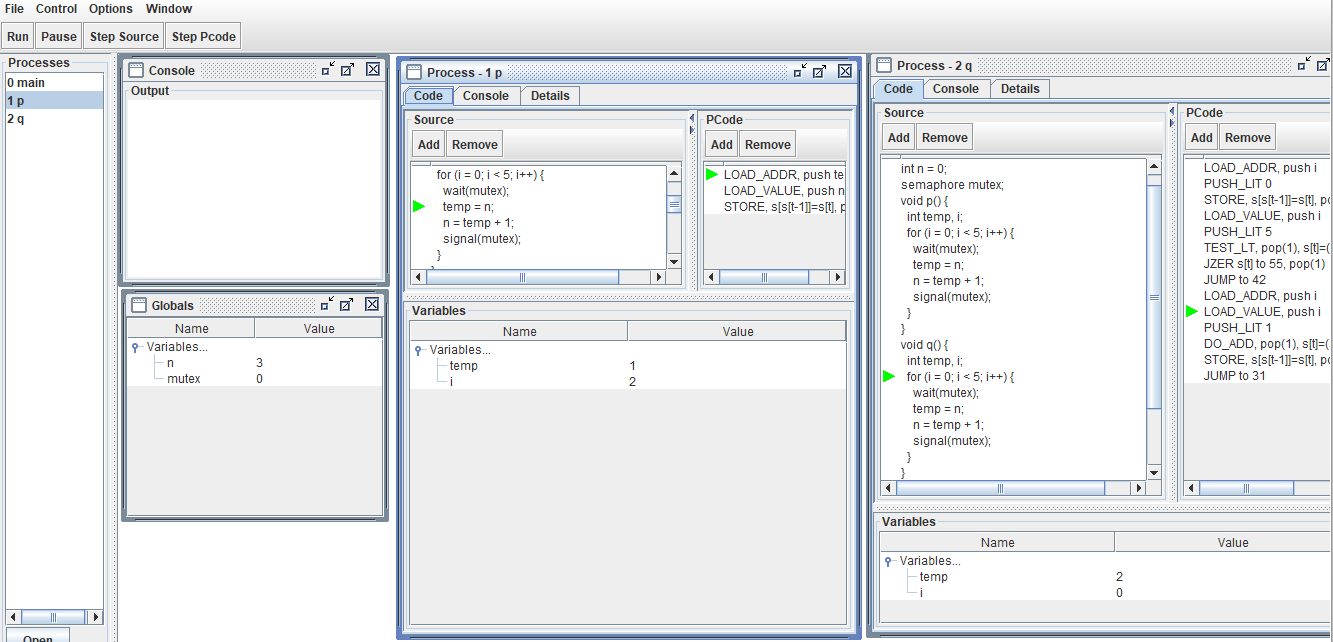


Ilustración 5 Ejecucuión código 2

## Código 3

El algoritmo de Dekker es un algoritmo de programación concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos.

Existen 5 versiones del algoritmo de dekker de las cuales la versión 5 es la que tiene mayor eficiencia y es tomado de la unión de la versión 1 y la versión 4.

|  |
| --- |
| Código 3 Dekker |
| |  | | --- | | int wantp = 0; | |  | int wantq = 0; | |  | int turn = 1; | |  |  | |  | void p() | |  | { | |  | while (1) | |  | { | |  | cout << "process p non-critical section\n"; | |  | wantp = 1; | |  | while (wantq){ | |  | if (turn == 2) | |  | { | |  | wantp = 0; | |  | while (!(turn == 1)); | |  | wantp = 1; | |  | } | |  | } | |  | cout << "process p critical section\n"; | |  | turn = 2; | |  | wantp = 0; | |  | } | |  | } | |  |  | |  | void q() | |  | { | |  | while (1) | |  | { | |  | cout << "process q non-critical section\n"; | |  | wantq = 1; | |  | while (wantp){ | |  | if (turn == 1) | |  | { | |  | wantq = 0; | |  | while (!(turn == 2)); | |  | wantq = 1; | |  | } | |  | } | |  | cout << "process q critical section\n"; | |  | turn = 1; | |  | wantq = 0; | |  | } | |  | } | |  |  | |  | main() { | |  | cobegin { | |  | p(); | |  | q(); | |  | } | |  | } | |

Tabla 3 Código 3

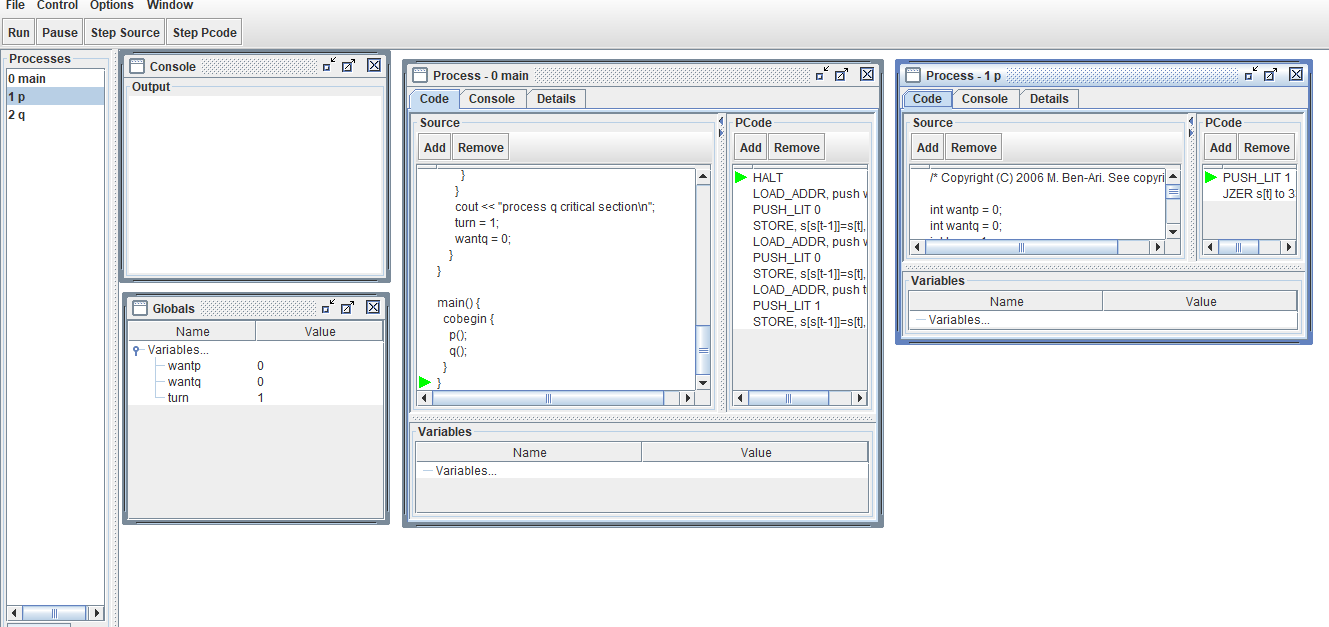


Ilustración 6 Ejecución código 3 Dekker

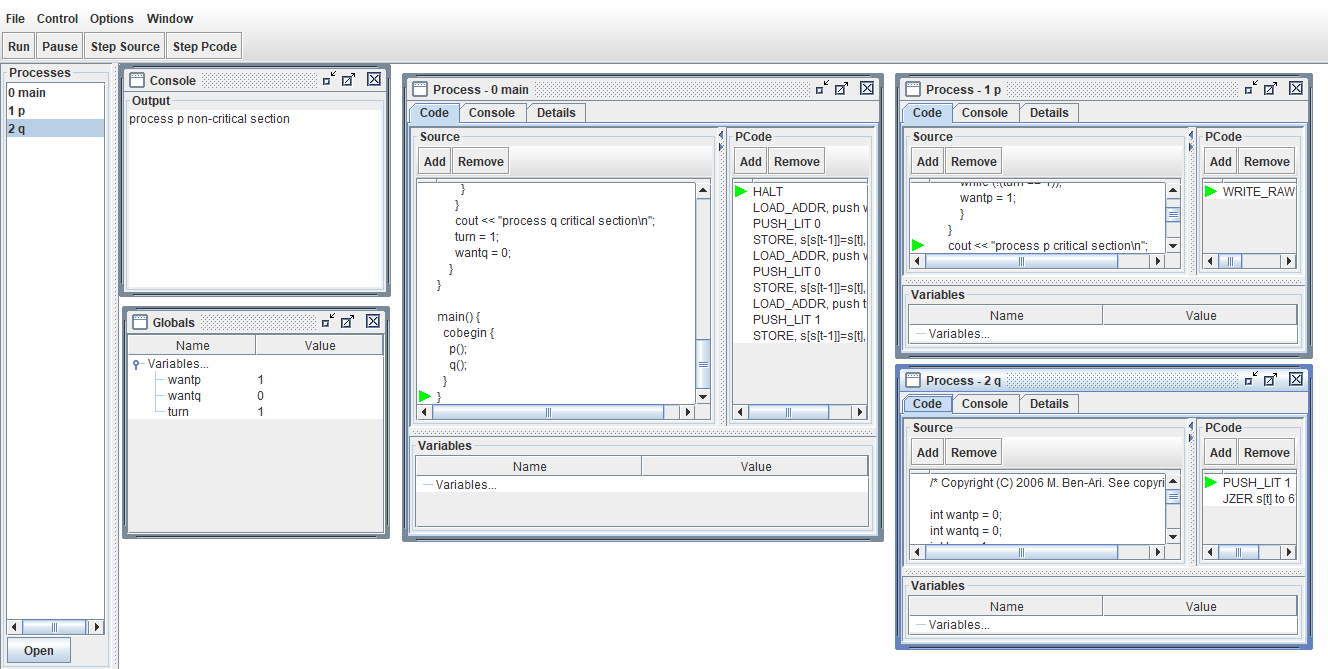


Ilustración 7 Algoritmo Dekker

El programa se ejecuta un cierto número de ocasiones, dónde espera el valor que tome la variable turn que indica el proceso que se ejecutará. Mientras la variable turn no cambie su valor, un proceso continua en ejecución hasta que se altera dicho valor y el siguiente proceso comienza su intervalo de ejecución.

## Código 4

|  |
| --- |
| Código 4 Dekker 1 |
| int wantp = 0;  int wantq = 0;  int turn = 1;  int N=0;  /\*{N=N\_{0}}p{N=N\_{0}+10}\*/  void p()  {  int i=0;  while (i<10)  {  cout << "process p non-critical section\n";  i++;  wantp = 1;  while (wantq){  if (turn == 2)  {  wantp = 0;  while (!(turn == 1));  wantp = 1;  }  }  cout << "process p critical section\n";  N=N+1;  turn = 2;  wantp = 0;  }  }  /\*{N=N\_{0}}q{N=N\_{0}+10}\*/  void q()  {  int i=0;  while (i<10)  {  cout << "process q non-critical section\n";  i++;  wantq = 1;  while (wantp){  if (turn == 1)  {  wantq = 0;  while (!(turn == 2));  wantq = 1;  }  }  cout << "process q critical section\n";  N=N+1;  turn = 1;  wantq = 0;  }  }  main() {  /\*{N=N\_{0}}cobegin{N=N\_{0}+20}\*/  cobegin {  p();  q();  }  } |

Tabla 4 Código Dekker 1

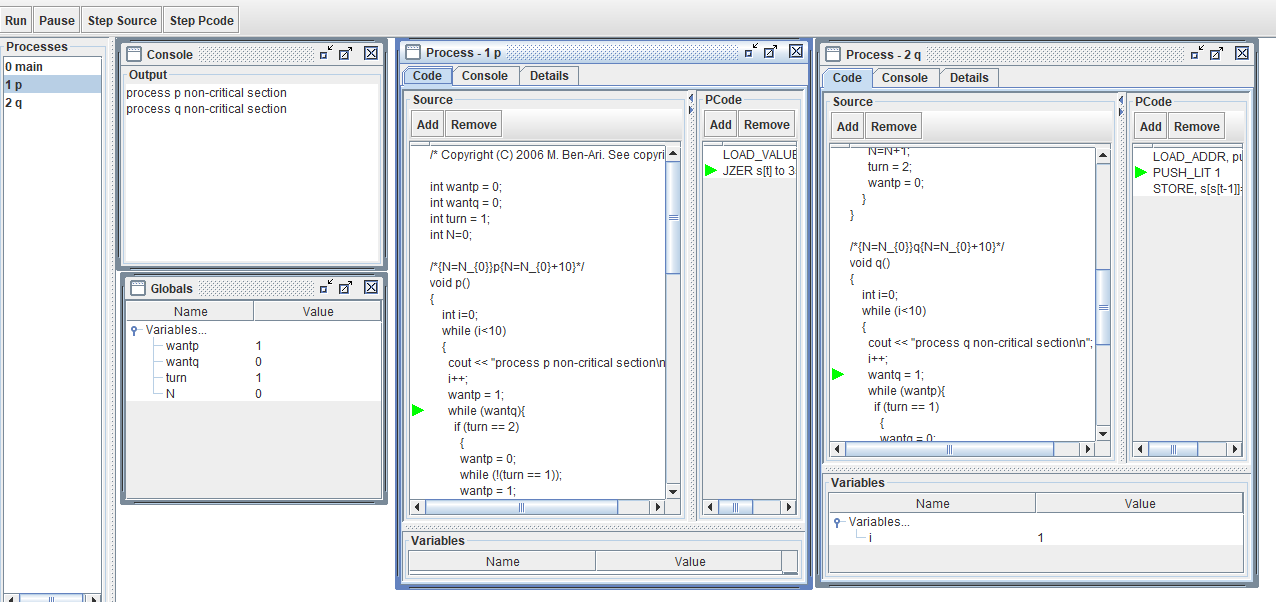


Ilustración 8 Ejecución del código en GNat

El programa se ejecuta un cierto número de ocasiones, dónde espera el valor que tome la variable turn que indica el proceso que se ejecutará. Mientras la variable turn no cambie su valor, un proceso continua en ejecución hasta que se altera dicho valor y el siguiente proceso comienza su intervalo de ejecución.

## Código 5

|  |
| --- |
| Código 5 Dekker 2 |
| int wantp = 0;  int wantq = 0;  int turn = 1;  int N=0;  int SALIR=0;  /\*{N=N\_{0}}p{N=N\_{0}+10}\*/  void p()  {  int i=0;  while (i<10)  {  cout << "process p non-critical section\n";  i++;  if(N>5){  cout<<"process p N="<<N<<endl;  for(;;){  if(N>10){  cout<<"Abandonamos el for infinito de p()\n";  SALIR=1;  break;  }  }  }  if(SALIR){  cout<<"Salimos del while de p()\n";  break;  }  wantp = 1;  while (wantq){  if (turn == 2)  {  wantp = 0;  while (!(turn == 1));  wantp = 1;  }  }  cout << "process p critical section\n";  N=N+1;  turn = 2;  wantp = 0;  }  }  /\*{N=N\_{0}}q{N=N\_{0}+10}\*/  void q()  {  int i=0;  while (i<10)  {  cout << "process q non-critical section\n";  i++;  wantq = 1;  while (wantp){  if (turn == 1)  {  wantq = 0;  while (!(turn == 2));  wantq = 1;  }  }  cout << "process q critical section\n";  N=N+1;  turn = 1;  wantq = 0;  }  }  main() {  /\*{N=N\_{0}}cobegin{N=N\_{0}+20}\*/  cobegin {  p();  q();  }  } |

Tabla 5 Código Dekker 2

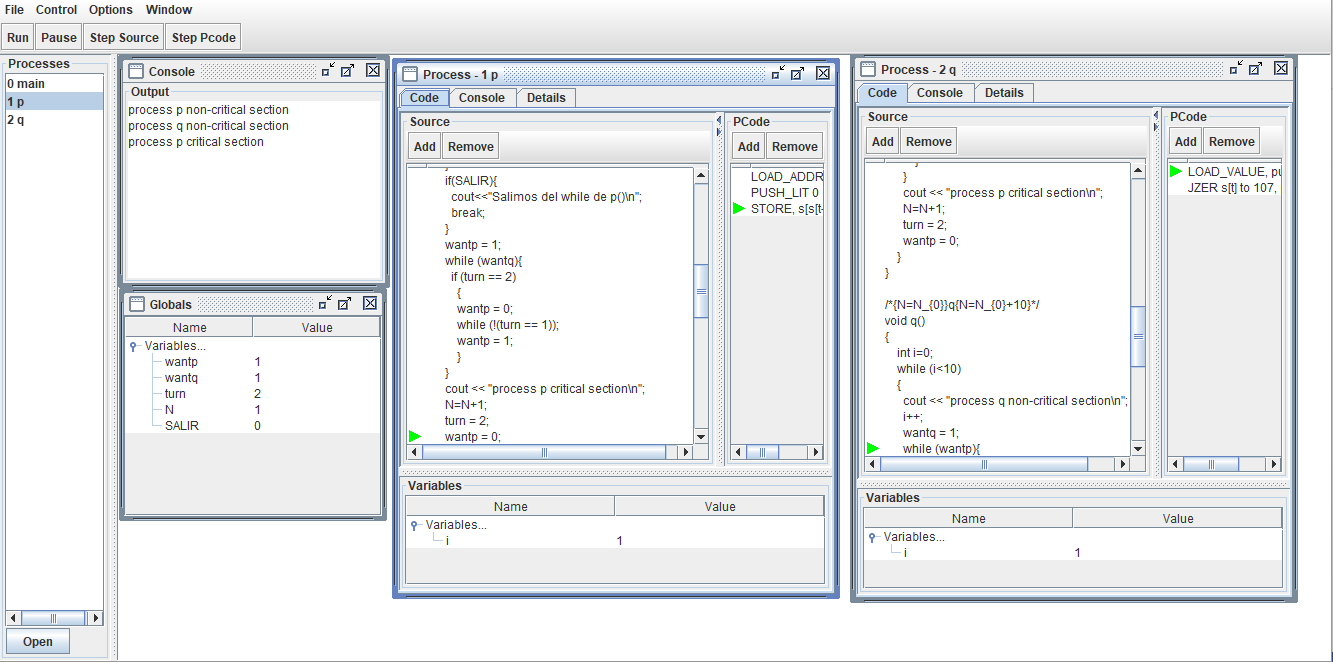


Ilustración 9 Ejecución código Dekker 2

## Código 6

El algoritmo de Peterson, es un algoritmo de programación concurrente para exclusión mutua, que permite a dos o más procesos o hilos de ejecución compartir un recurso sin conflictos, utilizando solo memoria compartida para la comunicación.

Peterson desarrolló el algoritmo como una simplificación del algoritmo de Dekker para dos procesos. Posteriormente este algoritmo fue generalizado para N procesos.

|  |
| --- |
| Código 6 Peterson |
| int FALSE=0;  int TRUE=1;  const int N=2; /\* Numero de procesos \*/  int turn; /\* ¿A quien le toca?\*/  int interested[N]; /\* Todos los valores son inicialmente 0 (FALSE) \*/  void enter\_region(int process) /\* process 0 o 1 \*/  {  int other; /\* Numero del otro proceso \*/  other=1-process; /\* Lo opuesto de process \*/  interested[process]=TRUE; /\* mostrar interes \*/  turn=process; /\*establecer bandera\*/  while((turn==process)&&(interested[other]==TRUE))/\* instruccion nula \*/;  }  void leave\_region(int process) /\* process: quien sale \*/  {  interested[process]=FALSE;  }  /\* process 0 \*/  void p()  {  while(1){  cout<<"p seccion no critica"<<endl;  enter\_region(0);  cout<<"p seccion critica"<<endl;  leave\_region(0);  }  }  /\* process 1 \*/  void q()  {  while(1){  cout<<"q seccion no critica"<<endl;  enter\_region(1);  cout<<"q seccion critica"<<endl;  leave\_region(1);  }  }  main() {  cobegin {  p();  q();  }  }/\* end main() \*/ |

Tabla 6 Código algoritmo Peterson

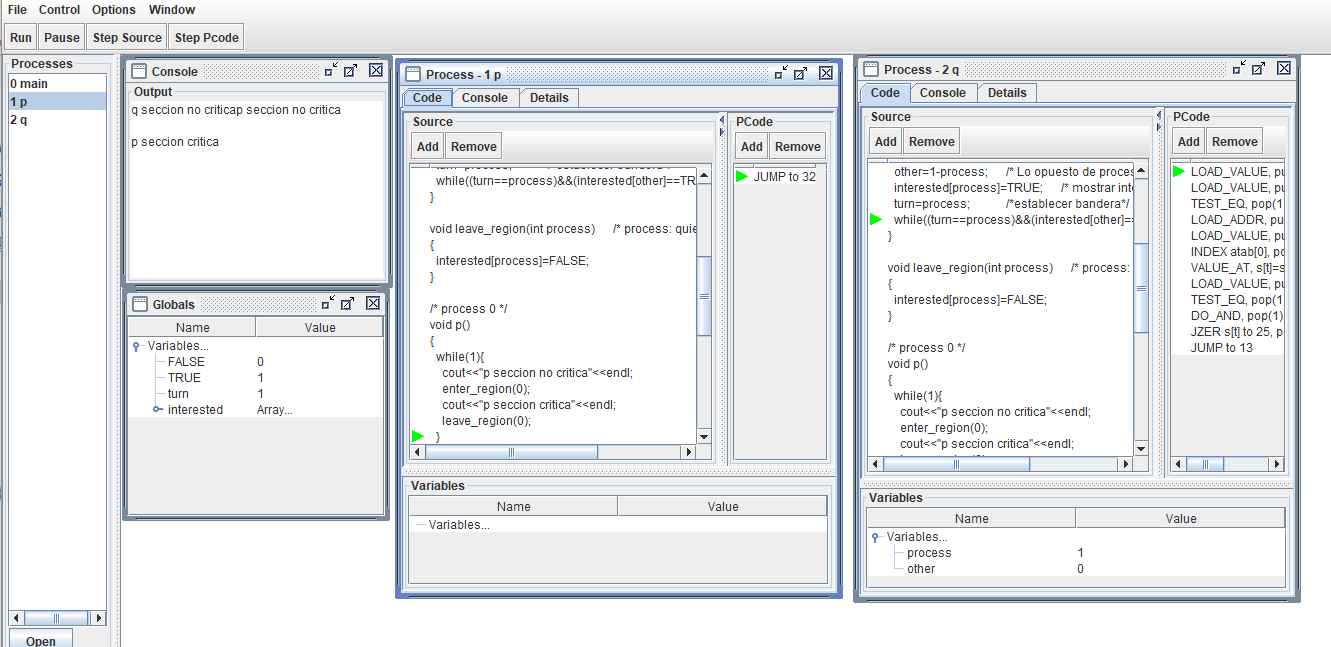


Ilustración 10 Ejecución algoritmo Peterson

# Conclusiones

Realizando la verificación del funcionamiento de los códigos empleados para la práctica se analizó el proceso de programación concurrente. Dónde se simulaba la ejecución de dos procesos simultáneamente, los cuales incrementaban el valor de una variable en un ciclo en ejecución que se repite un cierto número de ocasiones. Se pudo observar con ayuda del compilador baci de java, con su función console, como el proceso se ejecuta una vez cada que se utiliza la opción run para visualizar de manera más lenta como el valor de la variable va incrementando conforme se ejecuta cada proceso.