Práctica de Laboratorio
Procesos e hilos

1. Marco Teórico

1.1. Procesos

1.1.1. Ciclo de vida de un proceso

El ciclo de vida que suele seguir un proceso se puede ver en la Figura 1. El ciclo de vida en todos los Sistemas Operativos es estándar.

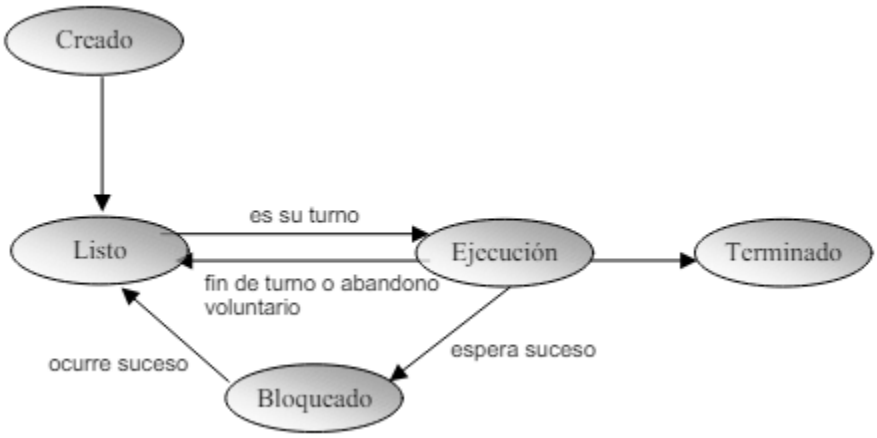


Figura 1. Estados de un proceso

1.1.2. Estados de un proceso

Creado: La forma de creación de un proceso depende del lenguaje del que se trate. Es el primer estado de su ciclo de vida. Inmediatamente después de creado pasa el estado de "Listo".

Listo: El proceso en estado listo espera que el planificador (parte del Sistema Operativo) le asigne tiempo de CPU. El proceso pasa a estado listo inmediatamente después de haber sido creado. También puede pasar a estado listo cuando su tiempo de uso de CPU expiró. Otra forma que un proceso llegue a estado de listo es que haya sido bloqueado previamente (a la espera de un evento) y cuando se produce el evento pasa a estado de listo.

En ejecución: Cuando el planificador asigna tiempo de CPU al proceso en estado de listo, este comienza a ejecutarse.

Bloqueado: Cuando un proceso está en ejecución pero para continuar debe ocurrir algún evento (liberación de recurso requerido, por ejemplo), es bloqueado y queda en ese estado hasta que ocurra el evento que está esperando, momento en el cual pasa a estado de listo.

Terminado: Cuando finaliza su ejecución.

Cambio de contexto: Es el cambio de estados de un proceso

1.1.3. Disposición en memoria de un proceso

En un Sistema Operativo (SO), la memoria se divide en espacio de usuario y espacio del núcleo.

En un SO multitarea, la información de un proceso queda dividida en dos espacios. Como se puede observar en la Figura 2.

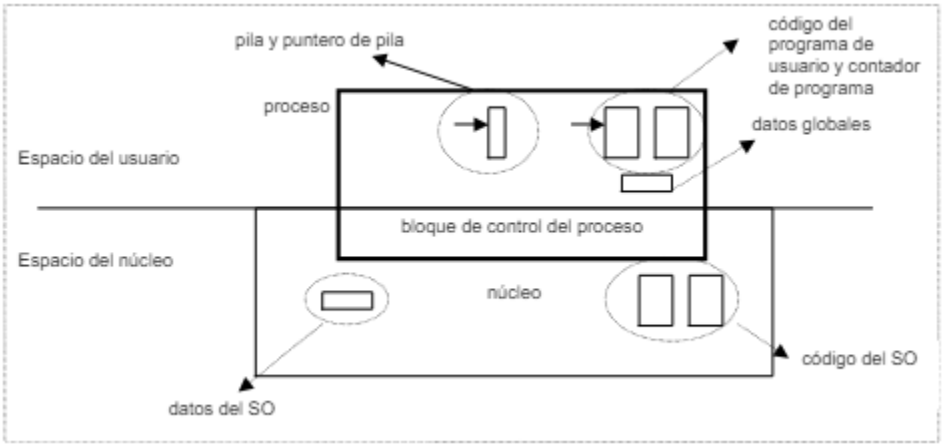


Figura 2. Mapa de memoria de un proceso para un SO multitarea

Espacio de usuario: Código del proceso, contador de programa, variables, pila y puntero de pila
Espacio de núcleo: Aquí reside el bloque de control de proceso o PCB con información del estado de proceso para los cambios de contexto. Los cambios de contexto son costosos desde el punto de vista del tiempo de ejecución.
Cuando tenemos más de un proceso, se tiene algo como lo representado en la Figura 3.

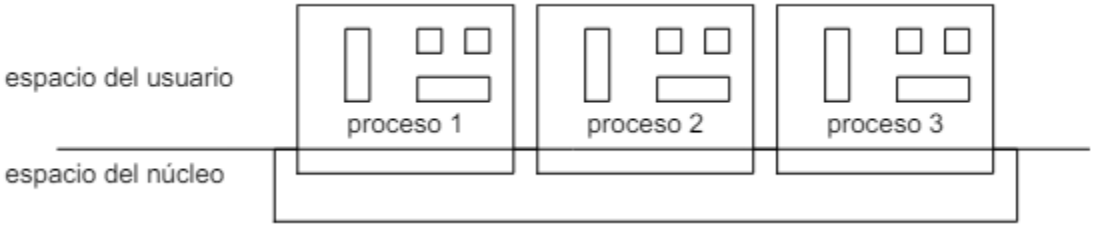


Figura 3. Varios procesos en un SO multitarea

1.1.4. Procesos en Pascal-FC

La estructura de un programa en Pascal-FC es:

```
program <identificador>
    /*Declaraciones globales*/
begin
    /*Sentencias*/
end.
```

Donde las declaraciones globales pueden ser:

- Constantes
- Tipos
- Variables
- Procedimientos

- Funciones
- Tipos de procesos
- Procesos
- Recursos
- Monitores

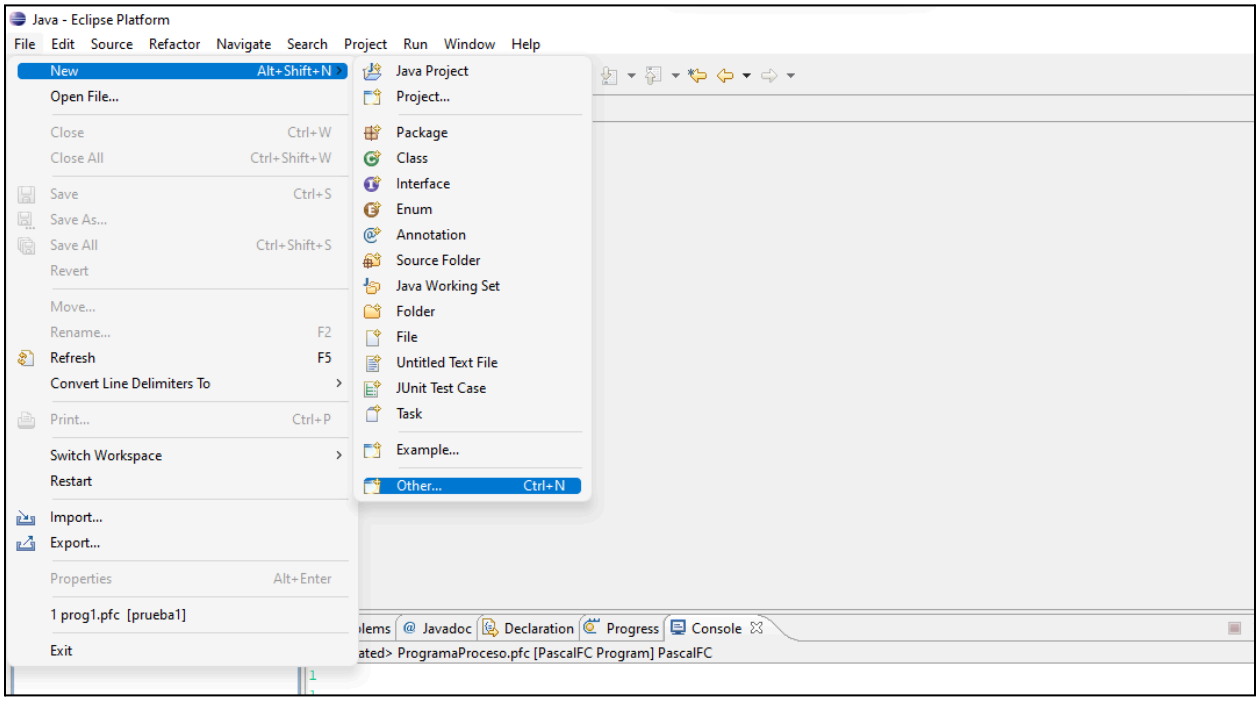
1.1.5. Pascal-FC en Eclipse-Gavab

Para la programación concurrente se usará la plataforma Eclipse Gavab.
Nuestro interés en esta plataforma radica en que se puede:

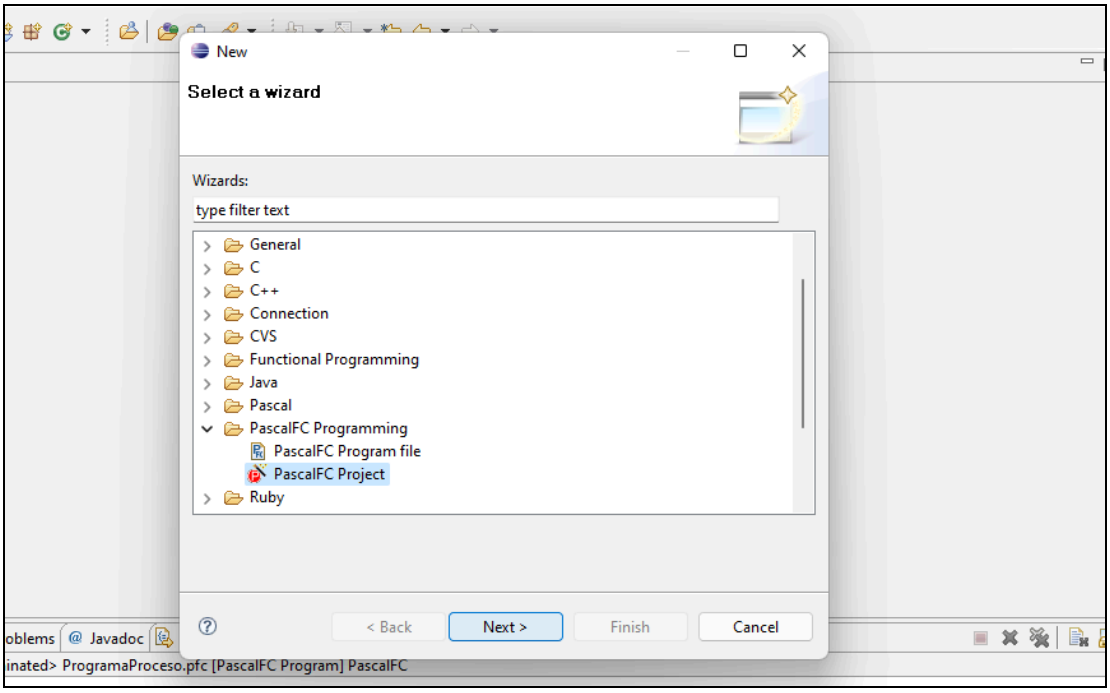
- Usar Pascal-FC para programación concurrente basada en múltiples procesos.

Creación de proyecto Pascal-FC

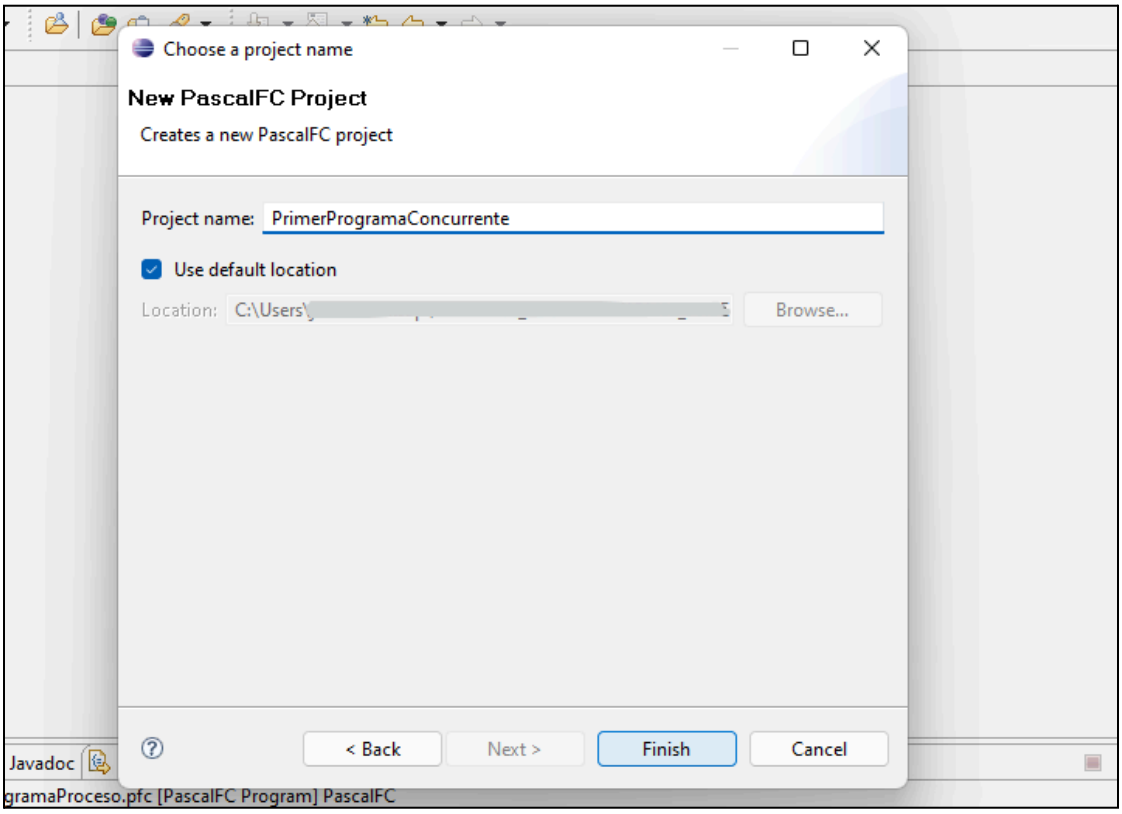
Para crear un proyecto Pascal-FC en Eclipse Gavab nos dirigimos a File>New>Other. Y seleccionamos la opcion "Other…"



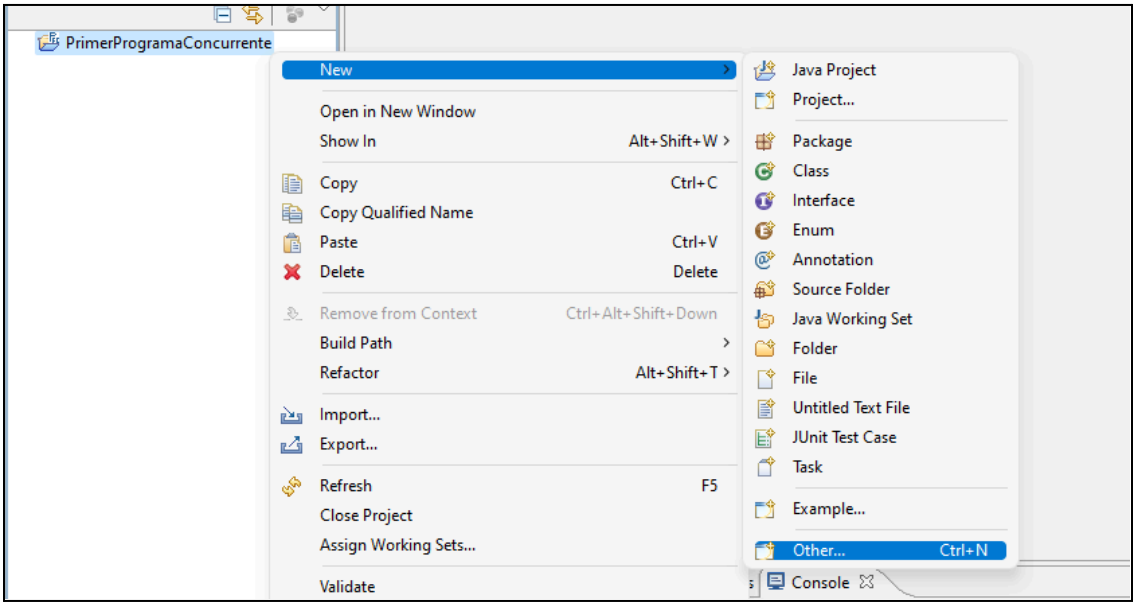
Luego, nos dirigimos a la carpeta PascalFC Programming y seleccionamos "PascalFC Project" y presionamos "Next >"



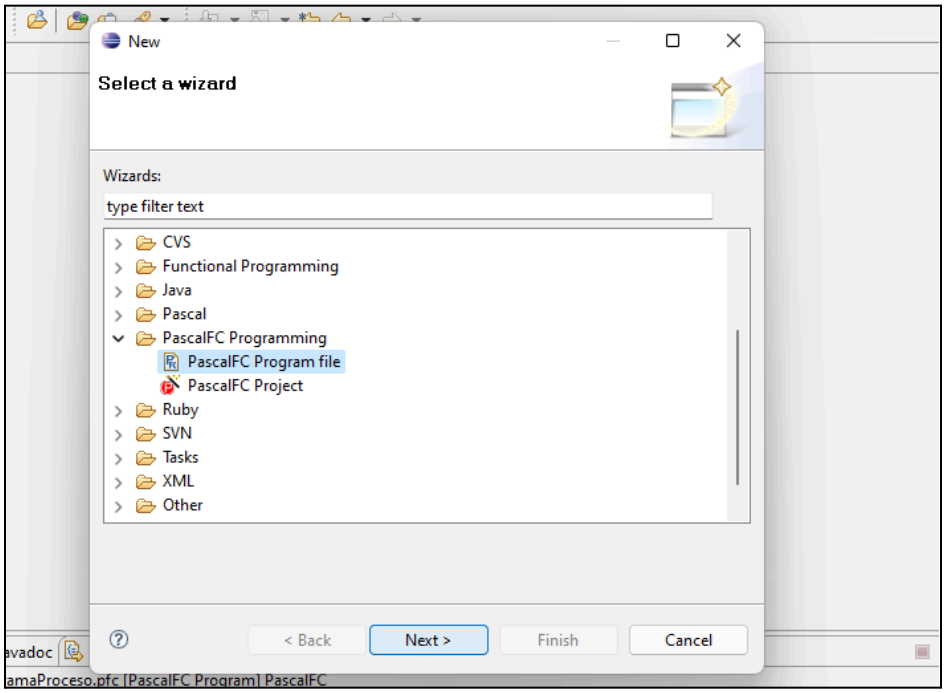
Luego escribimos el nombre de nuestro proyecto por ejemplo: “PrimerProgramaConcurrente” y seleccionamos “Finish”.



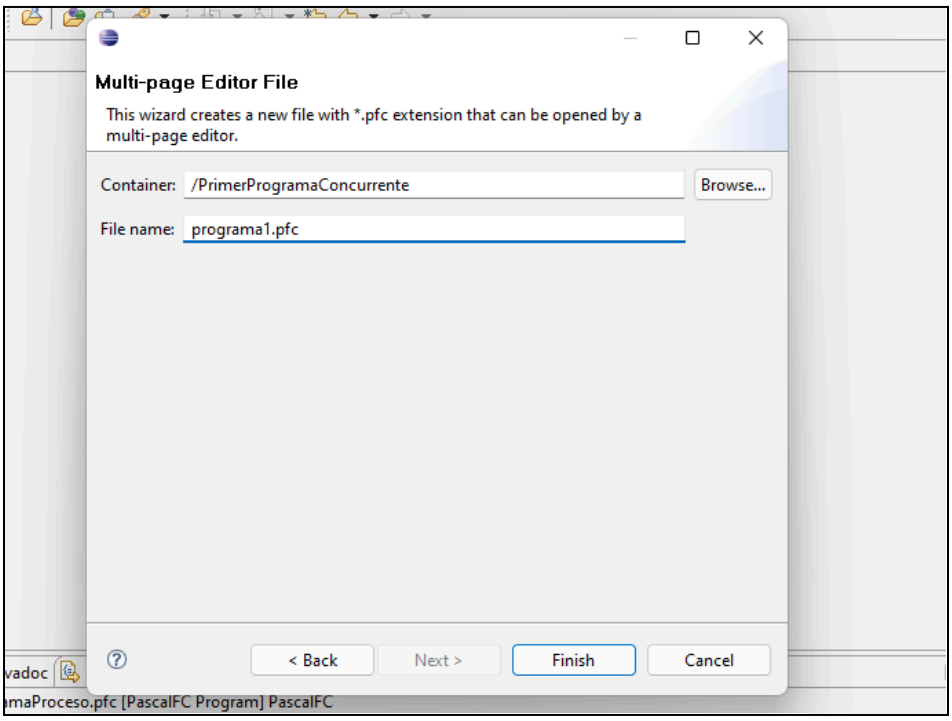
Veremos al lado derecho que se creó nuestro proyecto. Para crear un archivo, seleccionamos con click derecho sobre nuestro proyecto luego New>Other… Y seleccionamos “Other…”



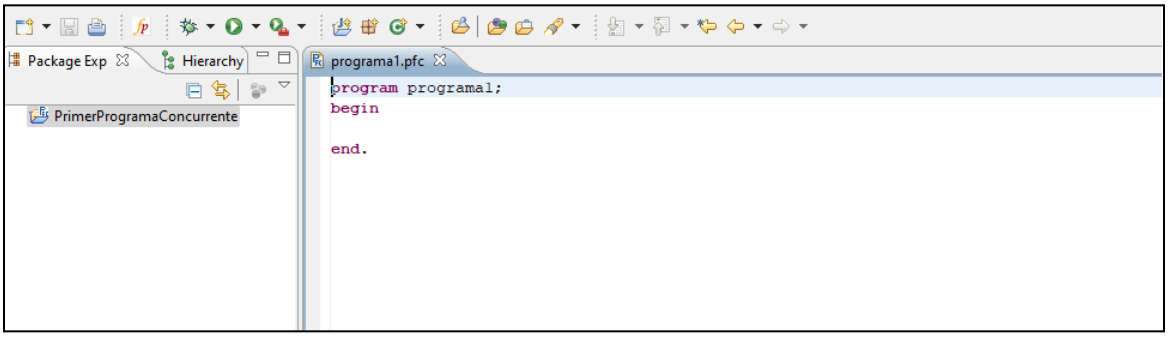
Seguidamente vamos a la carpeta PascalFC Programming y seleccionamos “PascalFC Program File”. Luego presionamos “Next >”.



A continuación, escribimos un nombre a nuestro archivo por ejemplo “programa1.pfc”. Y seleccionamos “Finish”.



Finalmente, se nos creará nuestra área de trabajo.



1.1.5. Declaración de procesos en PASCAL-FC:

Ejemplo 1:

Codificación

```
program UnoDosV1;
(*Declaraciones*)
var i,j:integer;

process Uno;
begin
    for i:=1 to 100 do
    begin
        writeln(1);
    end;
end;

process Dos;
begin
    for j:=1 to 100 do
    begin
        writeln(2);
    end;
end;
```

```
(*Var*)
(*Declaracion de variables globales*)
begin
    writeln('Inicio 1 y 2');
    cobegin
        Uno;
        Dos;
    coend;
    writeln('Final');
end.
```

Terminal:

```
Inicio 1 y 2
2
21
1
1


2
2
1
12
2
```

...

```
1
1
1
1
1
2
2
2
2
2
2
2
Final
```

Ejemplo 2:

Codificación:

```
program UnoDosV2;
(*Declaraciones*)
var i:integer;

process type Proceso (s:integer);
begin
    for i:=1 to 100 do
    begin
        writeln(s);
    end;
end;
```

```
var p1,p2: Proceso;
begin
    writeln('Inicio 1 y 2');
    cobegin
        p1(1);
        p2(2);
    coend;
    writeln('Final');
end.
```

Terminal:

```
Inicio 1 y 2
1
1
1
1
2
2
2
2
1
1
1
...
1
1
1
1
1
1
1
1
2
2
1
1
Final
```

1.1.6. Estados de un proceso en Pascal-FC

Respecto al ciclo de vida general que se vio para procesos, en Pascal-FC hay un cambio: el estado bloqueado cambia por otros tres: Esperando, Suspendido y TermEstado. Como se puede observar en la Figura 4.

Un proceso pasa al estado Esperando cuando se ejecuta la instrucción sleep (segundos). Esta instrucción deja al proceso durmiendo durante el lapso de tiempo especificado en el argumento, al despertar pasará al estado de listo.

Los estados Suspendido y TermEstado están relacionados con la espera selectiva en los mecanismos de paso de mensaje.

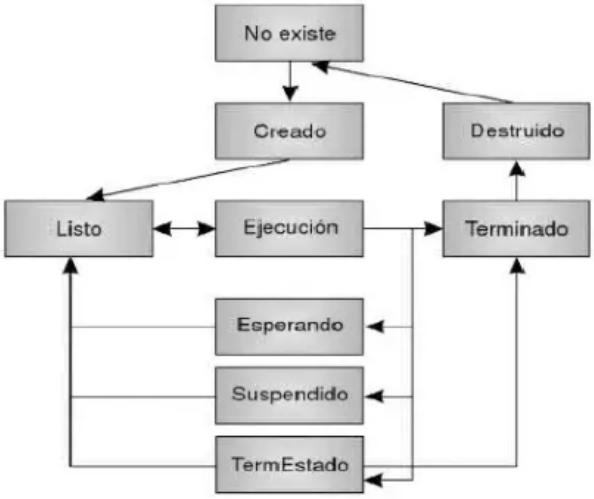


Figura 4. Los estados de un proceso en Pascal-FC

1.1.7. Planificación de procesos en Pascal-FC

Existen dos políticas de planificación en Pascal-FC configurables:

1. **Planificación injusta:** Ejecuta un proceso hasta que termine o se bloquee, entonces elegir otro proceso y ejecutarlo hasta que termine, y así sucesivamente con todos los procesos concurrentes, sin quitarles tiempo de CPU en ningún momento.
2. **Planificación justa:** Comparte el tiempo de procesador dando rodajas de tiempo a los procesos para su ejecución.

Se puede usar cualquiera de las dos planificaciones modificando un parámetro .
El eclipse Gavab está configurado para utilizar la planificación justa.

1.1.8. Ejercicio PascalFC

Se deben sumar el contenido de 4 matrices cuadradas.
Como el hardware subyacente posee 4 procesadores, se desea aprovechar esta capacidad para realizar la suma con un programa concurrente.
Desarrolle el programa en Pascal-FC para sumar las tres matrices.

Codificación

```
program SumaMatricesV1;
var matA,matB,matC,matSuma: array[1..4, 1..4] of integer;
    limSupFila, limInfFila, limSupCol, limInCol: integer;
(*Tipo proceso*)
process type ProcesoSumaMatrices(limInfFila,limSupFila,limInfCol,limSupCol: integer);
var i,j:integer;
begin
    for i:= limInfFila to limSupFila do
    for j:= limInfCol to limSupCol do
        begin
            matSuma[i,j] := matA[i,j] + matB[i,j] + matC[i,j];
        end;
end;

var p1,p2,p3,p4: ProcesoSumaMatrices;
i,j: integer;

begin
    for i:=1 to 4 do
    for j:=1 to 4 do
        begin
            matA[i,j] := 1;
            matB[i,j] := 2;
            matC[i,j] := 3;
            matSuma[i,j] := 0;
        end;
        cobegin
            p1(1,1,1,4);
            p2(2,2,1,4);
            p3(3,3,1,4);
            p4(4,4,1,4);
        coend;
    for i:=1 to 4 do
    for j:=1 to 4 do
        begin
            writeln('Elemento ',i,', ',j,' de la matriz Suma: ', matSuma[i,j]);
        end;
end.
```

Terminal

```
Elemento 1, 1 de la matriz Suma: 6
Elemento 1, 2 de la matriz Suma: 6
Elemento 1, 3 de la matriz Suma: 6
Elemento 1, 4 de la matriz Suma: 6
Elemento 2, 1 de la matriz Suma: 6
Elemento 2, 2 de la matriz Suma: 6
```

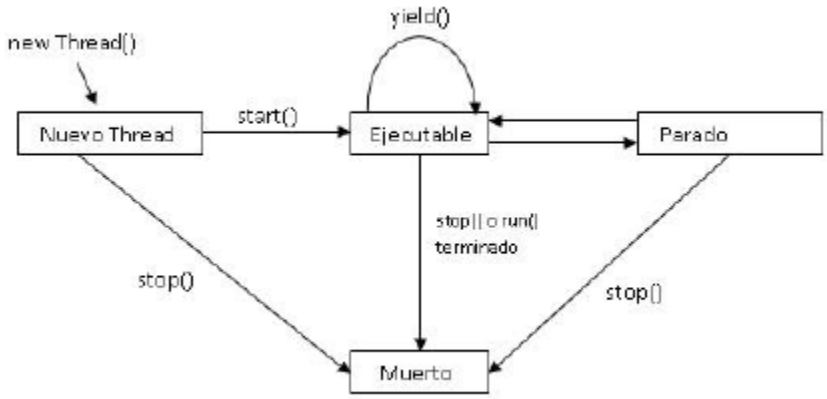
...

```
Elemento 3, 4 de la matriz Suma: 6
Elemento 4, 1 de la matriz Suma: 6
Elemento 4, 2 de la matriz Suma: 6
Elemento 4, 3 de la matriz Suma: 6
Elemento 4, 4 de la matriz Suma: 6
```

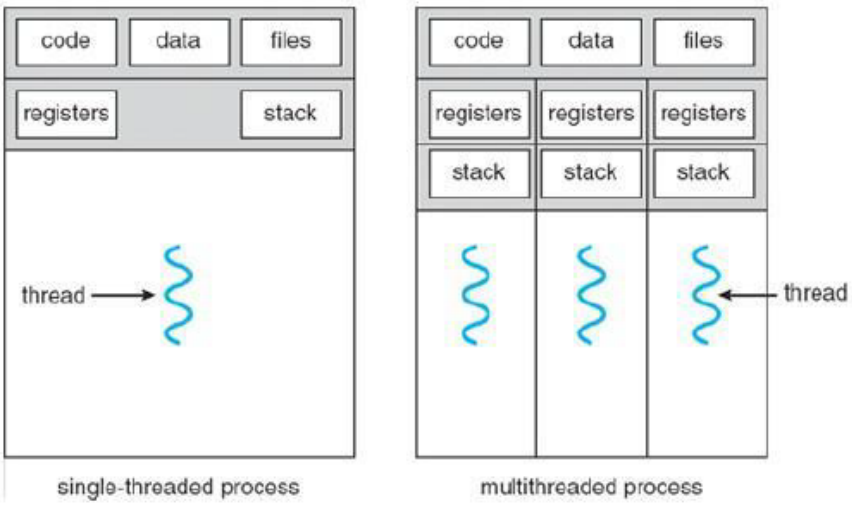
1.2. Hilos

Crear un nuevo hilo de ejecución (thread), definiendo estados:

- Nuevo: El primer estado de un hilo recién creado. Permanece en este estado hasta que el hilo es ejecutado.
- Ejecutable: Una vez ejecutado pasa a este estado, durante el cuál ejecuta su tarea.
- Parado – No ejecutable: Estado que permite al hilo desocupar la CPU en espera a que otro hilo termine o le notifique que puede continuar, o bien a que termine un proceso de E/S, o bien a que termine una espera provocada por la función Thread.sleep(100);. Tras ello volverá al estado Ejecutable.
- Muerto: Pasa a este estado una vez finalizado el método run().



Los hilos dentro de un proceso comparten toda la misma memoria, haciendo que si un hilo tocara una variable, todos los demás hilos del mismo proceso podrán ver el nuevo valor de la variable.



Un proceso es, por tanto, más costoso de lanzar, ya que se necesita crear una copia de toda la memoria de nuestro programa. Los hilos son más ligeros.

Elemento	PROCESO	THREAD
Espacio de direcciones	Le pertenece	Lo comparten
Stack/Pila	Le pertenece	Cada hilo tiene propio
Registros	Le pertenecen	Cada hilo tiene su propio conjunto
Archivos abiertos	Le pertenecen	Los comparten
Reloj	Le pertenece	Lo comparten
Variables globales	Le pertenecen	Las comparten
Hijos	Procesos hijos	Hilos hijos
Estado	Tiene propio	Cada hilo tiene el suyo
Protección entre	Existe	Usualmente no existe ya que cooperan entre sí
Señales	Le pertenecen	Las comparten

En resumen, las diferencias entre hilo y procesos radican en:

PROCESO	HILO
Son más pesados	Son más ligeros
Es cualquier programa en ejecución, independiente de otros procesos	Forman parte de un proceso, pueden haber varios hilos de ejecución
Un proceso acceden a su propia sección critica cuando son ejecutados a la vez, evitando que compartan memoria mediante exclusión mutua	Cuando un proceso hace varias cosas a la vez, los hilos dentro de un proceso comparten toda la misma memoria.
Los procesos necesitan crear una copia de toda la memoria de un programa	Si a un hilo le toca una variable, todos los demás hilos del mismo proceso verán el nuevo valor de la variable
Se usan procesos cuando se lanzan sus procesos hijos, no requieren demasiada comunicación con el mismo proceso, también para gestionar entradas y salidas	Se hacen uso de funciones para evitar que dos hilos a la vez accedan a la misma estructura de datos

1.3. Programando Hilos

En programación en C++ se usaron las siguientes instrucciones durante la ejecución de programas con hilos:

- Necesita de la librería “<thread>” en el encabezado del programa para comenzar a compilar los hilos.
- Para esperar ejecución de hilos, se usan operaciones con “join()”.
- Para interrumpir ejecución de hilos sin tener que esperar, se emplea “detach()”.
- Para identificar cada hilo, se emplea “get_id()”.
- Puede emplear operadores.

CASO 1:
Crear y ejecutar 3 hilos

```cpp
#include <iostream>
#include <thread>
#include <string>
#include <chrono>

using namespace std;

void saludo ( string m , int retardo , int veces ) {
    for ( int i =1; i <= veces ; i ++) {
        cout << m +"\n" ;
        this_thread :: sleep_for ( chrono :: milliseconds ( retardo ));
    }
}

int main()
{
  thread th_1(&saludo,"Mensaje A",100,10),
         th_2(&saludo,"Mensaje B",150,15),
         th_3(&saludo,"Mensaje C",300,5);
th_1.join ();
th_2.join ();
th_3.join ();
cout << "FIN !!" << endl;
    return 0;
}
```

CASO 2:
Crear y ejecutar un bucle de comunicación entre mensajes para un determinado hilo

```cpp
#include <iostream>
#include <thread>
#include <algorithm>
#include <vector>
#include <functional>
using namespace std;

void hacer_trabajo(unsigned id){
    cout << id << "\n";
}
void captura(int variable){
    vector<thread> hilos;
    for(unsigned i = 0; i <variable; i++){
      hilos.push_back(thread(hacer_trabajo,i));
    }

    for_each(hilos.begin(),hilos.end(), mem_fn(&thread::join));
}
int main(){
    int val;
    cout << "Introduce un numero";
    cin >> val;
    captura(val);
    return 0;
}
```

Convertimos miembro de una función del hilo a objeto usando mem_fn y este sea aplicado a cada uno de los elementos del vector

CASO 3

Ejecución por turnos en de forma simultánea.

```cpp
#include <iostream>
#include <thread>
#include <string>
#include <chrono>
using namespace std;

void saludo ( string m , int retardo , int veces ) {
    for ( int i =1; i <= veces ; i ++) {
        cout << m +"\n" ;
        this_thread :: sleep_for ( chrono :: milliseconds ( retardo ));
    }
}

int main(){
    thread P [3];
P [0] = thread(&saludo , "Mensaje A" , 100 , 10);
P [1] = thread(&saludo , "\tMensaje B" , 150 , 15) ,
P [2] = thread(&saludo , "\t\tMensaje C", 300 , 5);
    P [0].join();
    P [1].join();
    P [2].join();
cout << "FIN !!" << endl;
    return 0;
}
```

Se emplean vectores threads cuando se tienen varios hilos que ejecutan el mismo código

CASO 4

Usar constructores para la ejecución de hilos.

void comenzar_trabajo(p);

std::thread my_hilo(comenzar_trabajo,p);

```cpp
#include <iostream>
#include <thread>

using namespace std;

void function_1(string msg) {
        cout<< "Mensaje del hilo: " << msg<< "\n";
}

int main()
{
    std::thread t1(function_1, string("1"));

    try {
        for ( int i=0; i<100; i++ )
            cout << t1.get_id() << "Del menu principal " << i<< "\n";
    } catch(...) {
        t1.join();
        throw;
    }

    t1. join();

    return 0;
}
```

En caso de que suceda algo al hilo principal, los hilos hijos podrían terminar inesperadamente, se recomienda usar excepciones.

CASO 5

Usar destructores después de ejecutar hilo

```cpp
#include <iostream>
#include <chrono>
#include <thread>
using namespace std;
void independentThread(){
    std::cout << "Iniciando hilo concurrence.\n";
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Saliendo hilo concurrente.\n";
}
void threadCaller(){
    std::cout << "Iniciador de llamada de hilo.\n";
    std::thread t(independentThread);
    t.detach();
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Saliendo de llamada de hilo.\n";
}
int main(){
    threadCaller();
    std::this_thread::sleep_for(std::chrono::seconds(5));
    return 0;
}
```

Usa "detach()" para separar hilos cuando otro hilo esta ejecutándose.

CASO 6

Compartir valores entre hilos mediante clases

```
class repositorio_tareas {
public:
   void ejecutar(p){
      hacer_algo();
      hacer_otra_cosa();
    }
};
repositorio_tareas * f = new repositorio_tareas;
std::thread my_hilo(&repositorio_tareas::execute ,f,p);
```

```cpp
#include <iostream>
#include <thread>
using namespace std;
class Task
{
public:
    void execute(std::string command)
    {
        for(int i = 0; i < 5; i++)
        {
            std::cout<<command<<" :: "<<i<< "\n";
        }
    }
};

int main()
{
    Task * taskPtr = new Task();
    std::thread th(&Task::execute, taskPtr, "Emitir trabajo");
    th.join();
    delete taskPtr;
    return 0;
}
```

Crear y borrar un hilo usando instancia de clases y funciones

CASO 6

Usar funciones estáticas para ejecución y transmisión de hilos.

```cpp
#include <iostream>
#include <thread>
using namespace std;
class Task
{
public:
    static void test(std::string command)
    {
        for(int i = 0; i < 5; i++)
        {
            std::cout<<command<<" :: "<<i<<std::endl;
        }
    }
};

int main()
{
    std::thread th(&Task::test, "trabando con funciones estaticas: ");
    th.join();
    return 0;
}
```

Se crean hilos usando funciones estáticas

CASO 7

Emplear variables compartidas

```
class repositorio_tareas {
// colocar funcion
};
std::thread my_hilo(repositorio_tareas());
```

```cpp
#include <iostream>
#include <thread>
#include <string>
#include <chrono>

using namespace std;
// Parte de Especificaci´on : variables y funciones
// de los objetos de esta clase

class Saludador {
    public :
        Saludador(string mens , // constructor suministrando datos
                  int retardo ,
                  int veces );
    Saludador(): // constructor por defecto
    void run();
    // m´as funciones , si las hubiera
    string mens ;
    int retardo , veces ;
};
// Implementaci´on ( las funciones de los objetos de la clase )
Saludador :: Saludador ( string mens , int retardo , int veces ) {
    this -> mens = mens ; // mens : par´ametro de la funci´on
// this -> mens : variable del objeto
    this -> retardo = retardo ;
    this -> veces = veces ;
};

Saludador :: Saludador () {
    mens = "" ; // mens : par´ametro de la funci´on
// this -> mens : variable propia
    retardo = 0;
    veces = 0;
};

void Saludador :: run () {
    for ( int i =1; i <= veces ; i ++) {
        cout << mens + "\n ";
        this_thread :: sleep_for ( chrono :: milliseconds ( retardo ));
    }
};

int main(){
    // creacion de los objetos : constructor con datos
    Saludador s1("Mensaje 1 " , 100 , 10),
              s2("\tMensaje 2 " , 150 , 15),
              s3("\t\tMensaje 3 " , 10 , 40);
// creacion de los objetos : constructor por defecto
    Saludador s4;
    s4 = {"\t\t\tMensaje 4 " , 2, 12};
    cout <<" veces : " << s4.veces << "\n" ;

thread th_1 = thread (&Saludador :: run , s1 );
thread th_2 = thread (&Saludador :: run , s2 );
thread th_3 = thread (&Saludador :: run , s3 );
thread th_4 = thread (&Saludador :: run , s4 );
// esperar a que vayan acabando
th_1.join ();
th_2.join ();
th_3.join ();
th_4.join ();

    return 0;
}
```

Para procesos con comportamiento complejos, se requiere encapsular en una clase de objetos, crear tantas instancias como sea posible y usar un hilo para lanzar la función deseada

CASO 8

Exportar valores luego de ejecución de hilos

```
#include <iostream>
#include <fstream>
#include <thread>
using namespace std;

void generaarchivo(){
std::ofstream outfile("NombreArchivo.txt");
outfile << "este es el nombre del archivo \n";
outfile.close();
}
int main()
{
    std::thread nombre(generaarchivo);
    nombre.join();

    return 0;
}
```

Usamos la instrucción "outfile" y la librería "fstream" para generar un archivo y enviar contenido mediante hilos.

Asignación

Diseñar los algoritmos concurrentes en Codeblock C++.

Referencias

Palma Méndez, J. T., Garrido Carrera, M. C., Sánchez Figueroa, F., & Quesada Arencibia, A. (2003). *Programación concurrente*. Ediciones Paraninfo, SA.