

Universidad Tecnológica de Panamá

**Facultad de Ingeniería de Sistemas
Computacionales**

Lic. en Ingeniería de Software

Campus Víctor Levi Sasso



Asignatura

Aplicaciones para Ambientes Distribuidos

Tema

Parcial 1

Estudiante

Ernesto Crespo

8-929-1657

Profesor

Regis Rivera

Fecha de entrega

20 de mayo de 2024

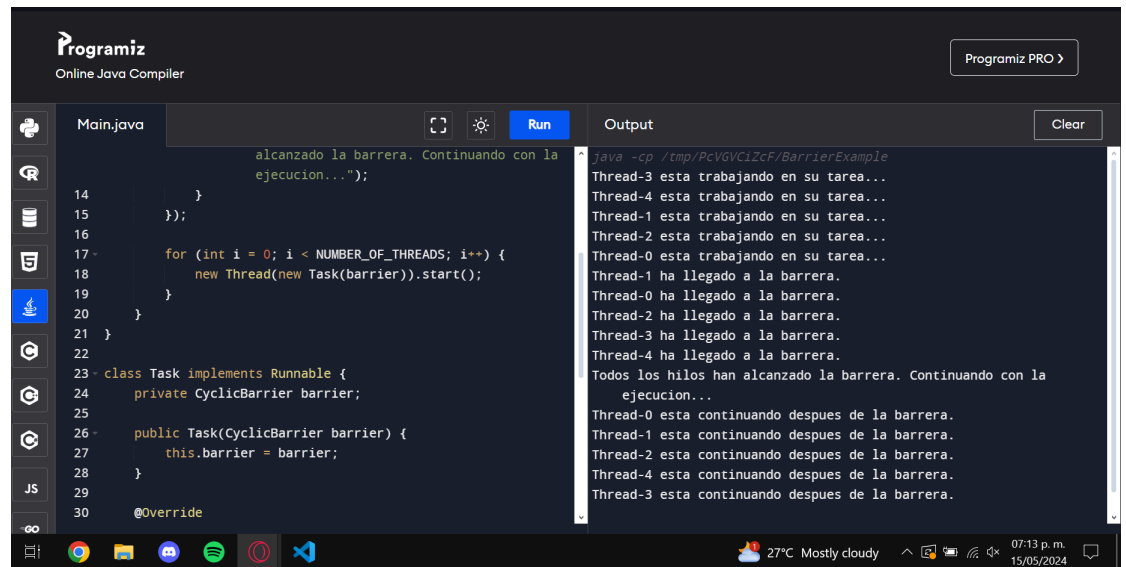
Desarrollo

Aplicaciones para Ambientes Distribuidos Parcial # 1 – Prof. Regis Rivera
Objetivo: Resolver los siguientes problemas de programación concurrente y paralela

I. Parte - Java

1. Crear programas que apliquen los siguientes semáforos

1.1 Barrier

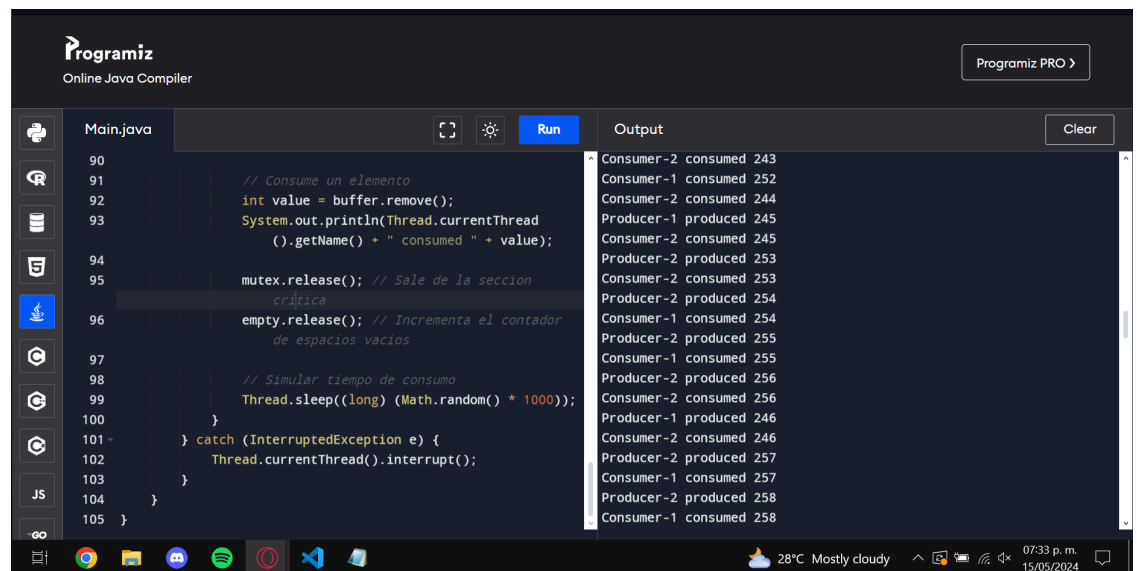


```
14      alcanzado la barrera. Continuando con la
15      ejecucion...");
16  });
17  for (int i = 0; i < NUMBER_OF_THREADS; i++) {
18      new Thread(new Task(barrier)).start();
19  }
20  }
21  }
22
23  class Task implements Runnable {
24      private CyclicBarrier barrier;
25
26      public Task(CyclicBarrier barrier) {
27          this.barrier = barrier;
28      }
29
30      @Override
```

```
java -cp /tmp/PcVGVc1ZcF/BarrierExample
Thread-3 esta trabajando en su tarea...
Thread-4 esta trabajando en su tarea...
Thread-1 esta trabajando en su tarea...
Thread-2 esta trabajando en su tarea...
Thread-0 esta trabajando en su tarea...
Thread-1 ha llegado a la barrera.
Thread-0 ha llegado a la barrera.
Thread-2 ha llegado a la barrera.
Thread-3 ha llegado a la barrera.
Thread-4 ha llegado a la barrera.
Todos los hilos han alcanzado la barrera. Continuando con la
ejecucion...
Thread-0 esta continuando despues de la barrera.
Thread-1 esta continuando despues de la barrera.
Thread-2 esta continuando despues de la barrera.
Thread-4 esta continuando despues de la barrera.
Thread-3 esta continuando despues de la barrera.
```

Explicación: Cada uno de los hilos realizará una tarea, una vez el hilo termine su tarea llamara a `barrier.await()` y entra en modo de espera. Cuando todos los hilos estén en modo de espera, la barrera se abrirá y todos los hilos continuarán su ejecución. La barrera los “retiene” hasta que todos los hilos estén en el `await()`.

1.2 Productores/consumidores



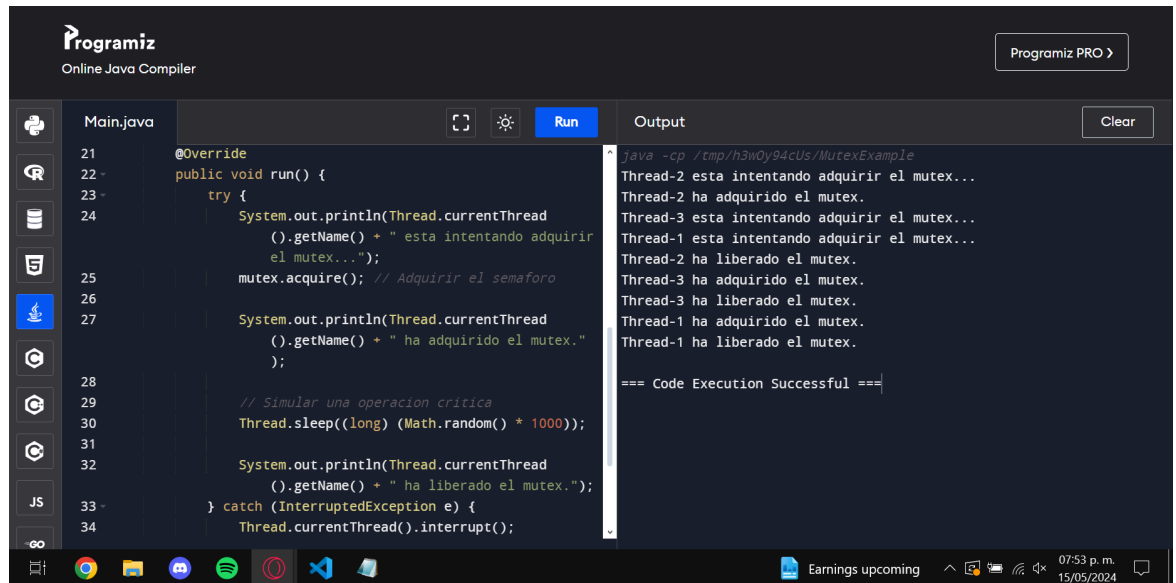
```
90
91      // Consume un elemento
92      int value = buffer.remove();
93      System.out.println(Thread.currentThread()
94          .getName() + " consumed " + value);
95
96      mutex.release(); // Sale de la seccion
97      // critica
98      empty.release(); // Incrementa el contador
99      // de espacios vacios
100
101      // Simular tiempo de consumo
102      Thread.sleep((long) (Math.random() * 1000));
103  } catch (InterruptedException e) {
104      Thread.currentThread().interrupt();
105  }
106  }
```

```
Consumer-2 consumed 243
Consumer-1 consumed 252
Consumer-2 consumed 244
Producer-1 produced 245
Consumer-2 consumed 245
Producer-2 produced 253
Consumer-2 consumed 253
Producer-2 produced 254
Consumer-1 consumed 254
Producer-2 produced 255
Consumer-1 consumed 255
Producer-2 produced 256
Consumer-2 consumed 256
Producer-1 produced 246
Consumer-2 consumed 246
Producer-2 produced 257
Consumer-1 consumed 257
Producer-2 produced 258
Consumer-1 consumed 258
```

Explicación: Los productores intentan producir elementos continuamente, si el buffer está lleno, esperaran hasta que haya espacio disponible.

Los consumidores intentan consumir elementos continuamente, si el buffer está vacío, esperará hasta que hayan elementos disponibles

1.3 Mutex



```
21 @Override
22 public void run() {
23     try {
24         System.out.println(Thread.currentThread
25             ().getName() + " esta intentando adquirir
26             el mutex...");
27         mutex.acquire(); // Adquirir el semaforo
28
29         System.out.println(Thread.currentThread
30             ().getName() + " ha adquirido el mutex."
31             );
32
33         // Simular una operacion critica
34         Thread.sleep((long) (Math.random() * 1000));
35
36         System.out.println(Thread.currentThread
37             ().getName() + " ha liberado el mutex.");
38     } catch (InterruptedException e) {
39         Thread.currentThread().interrupt();
40     }
41 }
```

Output:

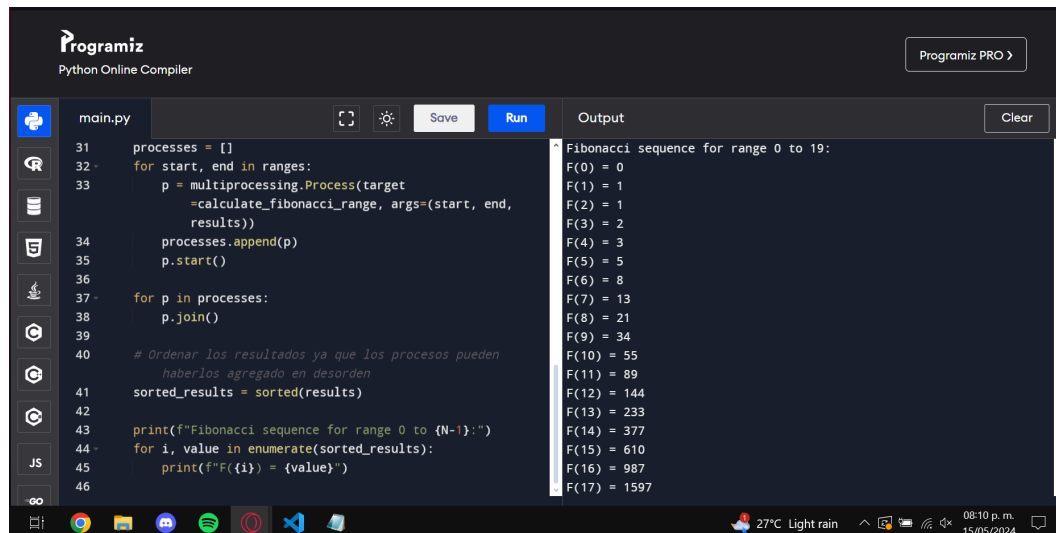
```
java -cp /tmp/h3w0y94cUs/MutexExample
Thread-2 esta intentando adquirir el mutex...
Thread-2 ha adquirido el mutex.
Thread-3 esta intentando adquirir el mutex...
Thread-1 esta intentando adquirir el mutex...
Thread-2 ha liberado el mutex.
Thread-3 ha adquirido el mutex.
Thread-3 ha liberado el mutex.
Thread-1 ha adquirido el mutex.
Thread-1 ha liberado el mutex.
=== Code Execution Successful ===
```

Explicación: Cuando se llama a “mutex.acquire()”, intenta adquirir el semáforo. Si el semáforo ya está adquirido por otro hilo, el hilo actual espera hasta que el semáforo esté disponible.

Después de completar la operación crítica, el hilo libera el semáforo llamando a “mutex.release()”, permitiendo así que otros hilos puedan adquirir el semáforo y acceder a la sección crítica.

II. Python

2. Crear un programa en Python que calcule la secuencia Fibonacci de un rango de N números utilizando varios procesos.



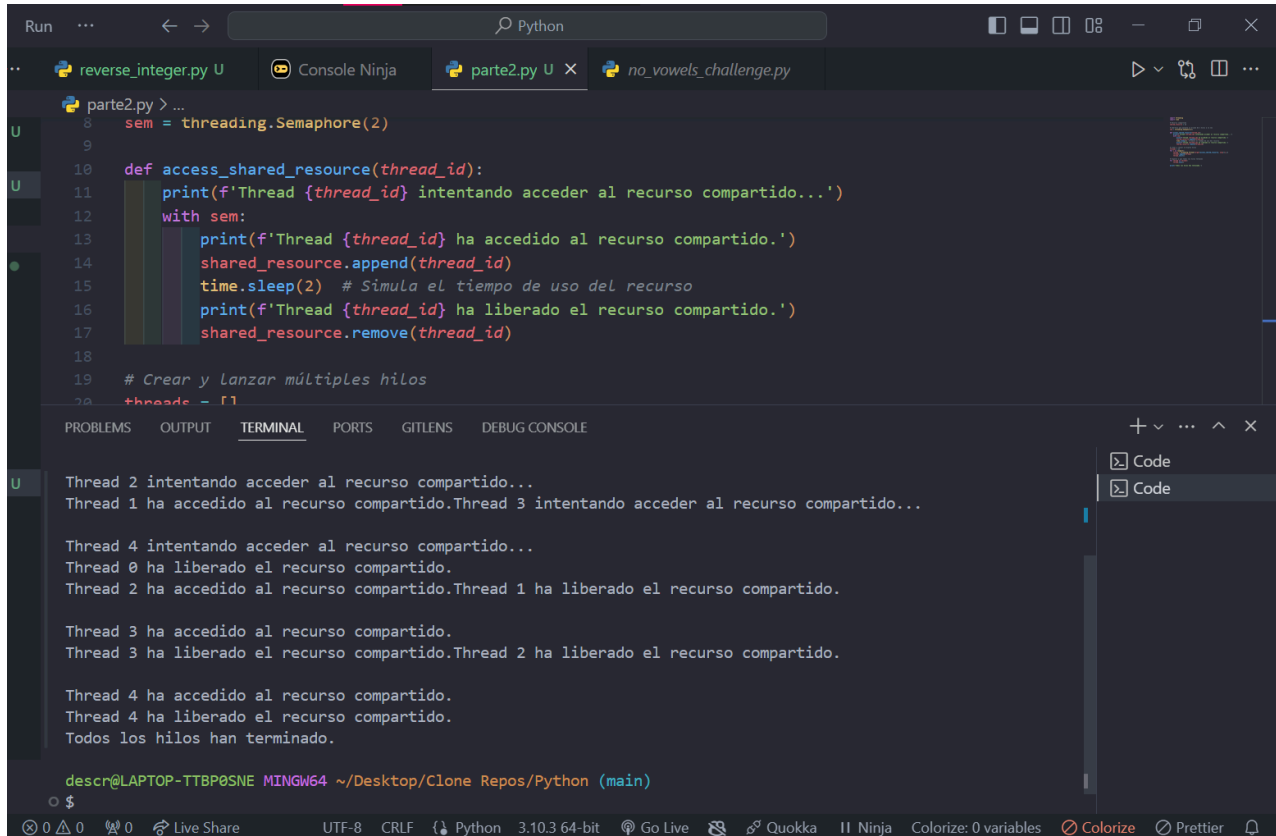
```
31 processes = []
32 for start, end in ranges:
33     p = multiprocessing.Process(target
34         =calculate_fibonacci_range, args=(start, end,
35         results))
36     processes.append(p)
37     p.start()
38
39 for p in processes:
40     p.join()
41
42 # Ordenar los resultados ya que los procesos pueden
43 haberlos agregado en desorden
44 sorted_results = sorted(results)
45
46 print(f"Fibonacci sequence for range 0 to {N-1}:")
47 for i, value in enumerate(sorted_results):
48     print(f"F({i}) = {value}")
```

Output:

```
Fibonacci sequence for range 0 to 19:
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
F(10) = 55
F(11) = 89
F(12) = 144
F(13) = 233
F(14) = 377
F(15) = 610
F(16) = 987
F(17) = 1597
```

Explicación: El programa calculará el n-ésimo número de la secuencia de Fibonacci en la iteración para evitar problemas de recursión y mejor rendimiento. Esta secuencia tiene un rango específico de 0 a 19

3. Crear un programa en Python que utilice un semáforo para controlar el acceso a un recurso compartido, con el fin de garantizar que más de 2 hilos accedan a un recurso compartido al mismo tiempo.



```
Run ... Python
reverse_integer.py U Console Ninja parte2.py U X no_vowels_challenge.py
parte2.py > ...
8 sem = threading.Semaphore(2)
9
10 def access_shared_resource(thread_id):
11     print(f'Thread {thread_id} intentando acceder al recurso compartido...')
12     with sem:
13         print(f'Thread {thread_id} ha accedido al recurso compartido.')
14         shared_resource.append(thread_id)
15         time.sleep(2) # Simula el tiempo de uso del recurso
16         print(f'Thread {thread_id} ha liberado el recurso compartido.')
17         shared_resource.remove(thread_id)
18
19 # Crear y lanzar múltiples hilos
20 threads = []

PROBLEMS OUTPUT TERMINAL PORTS GITLENS DEBUG CONSOLE
Thread 2 intentando acceder al recurso compartido...
Thread 1 ha accedido al recurso compartido.Thread 3 intentando acceder al recurso compartido...

Thread 4 intentando acceder al recurso compartido...
Thread 0 ha liberado el recurso compartido.
Thread 2 ha accedido al recurso compartido.Thread 1 ha liberado el recurso compartido.

Thread 3 ha accedido al recurso compartido.
Thread 3 ha liberado el recurso compartido.Thread 2 ha liberado el recurso compartido.

Thread 4 ha accedido al recurso compartido.
Thread 4 ha liberado el recurso compartido.
Todos los hilos han terminado.

descr@LAPTOP-TTB0SNE MINGW64 ~/Desktop/Clone Repos/Python (main)
$
```

Explicación: Importamos dos librerías “threading” y “time”, luego definimos “shared_resource” como lista de recursos compartidos. Creamos el semáforo llamado “sem” que tiene como máximo 2 hilos a la vez. Cada hilo intenta acceder al recurso compartido mediante la función “access_shared_resource”. El hilo simulara el uso del recurso añadiendo su ID en “shared_resource” y durmiendo 2 segundos, luego de usar el resource, el hilo eliminará su ID de “shared_resource”. Al final imprimimos un mensaje indicando que todos los hilos han terminado.

III. Lenguaje libre (el que usted desee)

4. Hoy es muy común crear aplicaciones con concurrencia y/o paralelas aprovechando todas sus ventajas, sin embargo ¿Cómo se pueden usar especialmente en el dominio de las aplicaciones web?

- i. En las aplicaciones web, gran parte de la concurrencia es administrada por el propio servidor web y muy rara vez veo múltiples subprocesos implementados dentro de las páginas web. AJAX también habilitó el paradigma tipo "pagelets" para ayudar aún más.
- ii. Las aplicaciones web normalmente consisten en obtener resultados rápidamente y hasta ahora utilizamos muchas tácticas como almacenamiento en caché, redundancia, etc. para lograr este objetivo. Si había algo que requería un uso intensivo de computación, tenía que suceder fuera de línea (y los clientes podían consultar los resultados más tarde o se podían implementar devoluciones de llamada).
- iii. En la mayoría de los casos, y en particular para aplicaciones web con uso intensivo, probablemente no sea necesario introducir paralelismo adicional, ya que agregar más elementos de trabajo sólo generará competencia por el tiempo de CPU y, en última instancia, reducirá el rendimiento de las solicitudes.
- iv. Las aplicaciones web que necesitan realizar cálculos costosos aún pueden beneficiarse del paralelismo si la latencia de una solicitud individual es más importante que el rendimiento general de la solicitud.
 - Basado en este contexto: Elabore una aplicación web que mejore un cálculo de un proceso en particular que si requiere aplica concurrencia y/o paralelismo más allá del alcance que ya de por si el servidor web ofrece
 - Utilice el lenguaje de su preferencia
 - Realice una aplicación web sencilla, es decir, no enfocarse en el front-end sino en el back-end (en si, lo solicitado). Si se crea en fondo blanco y texto en negro, con la fuente default (Times New Roman, Calibrí o la que designe el web browser, es suficiente)

```
reverse_integer.py Console Ninja parte2.py M parte3.py U x no_vowels_challenge.py
parte3.py > ...
15 class RequestHandler(BaseHTTPRequestHandler):
23     def do_POST(self):
33         # Enviar la respuesta
34         self._set_headers() # Establecer Los encabezados de la respuesta
35         self.wfile.write(json.dumps(results).encode('utf-8')) # Enviar Los resultados como JSON
36
37 # Define la función que se ejecutara en el servidor
38 def run(server_class=HTTPServer, handler_class=RequestHandler, port=8080):
39     server_address = ('', port) # Configurar la dirección del servidor
40     httpd = server_class(server_address, handler_class) # Crear una instancia del servidor HTTP
41     print(f'Starting httpd server on port {port}') # Imprimir mensaje de inicio del servidor
42     httpd.serve_forever() # Iniciar el bucle de servicio del servidor
43
44 # Comprueba que el script se ejecute como el programa principal
45 if __name__ == "__main__":
46     run() # Ejecutar la función run para iniciar el servidor
47
PROBLEMS OUTPUT TERMINAL PORTS GITLENS DEBUG CONSOLE
descr@LAPTOP-TTBP0SNE MINGW64 ~/Desktop/Clone Repos/Python (main)
$ python parte3.py
Starting httpd server on port 8080
127.0.0.1 - - [20/May/2024 16:53:25] "POST / HTTP/1.1" 200 -
[]
main* Launchpad 0 0 0 0 Live Share UTF-8 C
MINGW64/c/Users/descr
descr@LAPTOP-TTBP0SNE MINGW64 -
$ curl -X POST http://127.0.0.1:8080 -H "Content-Type: application/json" -d '{"numbers": [1,2,3,4,5]}'
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 41 0 17 100 24 4 5 0:00:04 0:00:04 --:--:-- 10[1, 4, 9, 16, 25]
descr@LAPTOP-TTBP0SNE MINGW64 -
$
```

En la captura anterior se muestra cómo enviamos una solicitud POST a nuestro servidor en el puerto 8080 con el siguiente comando: `curl -X POST http://127.0.0.1:8080 -H "Content-Type: application/json" -d '{"numbers": [1,2,3,4,5]}'`

Esto nos dará el resultado de la entrega al servidor, son su fecha y código de estado 200 en este caso. El código de esta parte está adjunto en la carpeta con los demás, su ruta es, Parte 3 (Folder), dentro el archivo llamado Backend.py.

IV. C++

5. Considere los siguientes dos programas

Programa 1

```
#include <cstdio>
#include <unistd.h>

int main() {
    printf("PID %d running prog1\n", getpid());
}
```

Programa 2

```
#include <cstdio>
#include <unistd.h>

int main() {
    char* argv[2];
    argv[0] = (char*) "prog1";
    argv[1] = nullptr;
    printf("PID %d running prog2\n", getpid());
    int r = execl("./prog1", argv);
    printf("PID %d exiting from prog2\n", getpid());
}
```

¿Cuántos PID diferentes se imprimirán si ejecuta el Programa 2?

-Solo un PID. El PID de proceso que ejecuta "prog2" en el mismo proceso que ejecutará "prog1" luego del "execl", ya que "execl" reemplaza el proceso actual en el mismo espacio de memoria.

¿Cuántas líneas de salida saldrán?

-Saldrán dos líneas de salida, primero "prog2" que imprimirá "PID [pid] running prog2"

Luego si “execv” tiene éxito, “prog1” imprime “PID [pid] running prog1”

Si “execv” es exitoso, no se ejecutarán las líneas después de “execv” en “prog2”

6. Ahora, si cambiamos el programa 2 por:

```
#include <stdio>
#include <unistd.h>
int main() {
    char* argv[2];
    argv[0] = (char*) "prog1";
    argv[1] = nullptr;

    printf("PID %d running prog2\n", getpid());
    pid_t p = fork();
    if (p == 0) {
        int r = execv("./prog1", argv);
    } else {
        printf("PID %d exiting from prog2\n", getpid());
    }
}
```

¿Cuántos PID diferentes se imprimirán si ejecuta el Programa 2B?

-Dos PID distintos, uno para el proceso del padre y otro para el proceso del hijo

¿Cuántas líneas de salida saldrán?

-Tres líneas de salida:

la primera línea de “prog2” en el proceso padre antes del “fork”: “PID [pid] running prog 2”

la segunda línea de “prog2” en el proceso padre después del “fork”: “PID [pid] exiting from prog2”

la línea de “prog1” en el proceso hijo después de “execv”: “PID [pid] running prog1”

7. Finalmente, considere esta otra versión del programa 2

```
#include <stdio>
#include <unistd.h>
int main() {
    char* argv[2];
    argv[0] = (char*) "prog1";
    argv[1] = nullptr;

    printf("PID %d running prog2\n", getpid());
    pid_t p = fork();
    pid_t q = fork();
    if (p == 0 || q == 0) {
        int r = execv("./prog1", argv);
    } else {
        printf("PID %d exiting from prog2\n", getpid());
    }
}
```

¿Cuántos PID diferentes se imprimirán si ejecuta el Programa 2C?

-Aquí se imprimirán 4 PID Distintos, uno del proceso original y 3 de procesos hijos creados por dos llamadas a “fork” en pid_t q y pid_t p.

¿Cuántas líneas de salida saldrán?

-Esto depende si “execv” tiene éxito o no:

Si tiene éxito, cada uno de los tres procesos hijos imprimirá: “PID [pid] running prog1”

El proceso original y uno de los procesos hijos que no llamo a “execv” imprime:

“PID [pid] exiting from prog2” dando un total de 4 líneas de salida.