PRACTICA 3

Objetivo General

Retomar los conocimientos de generación de Procesos aprendido en el curso de Programación Concurrente, pero ahora con una nueva herramienta de programación, Python.

Objetivos Particulares:

- Realizar un par de programas para verificar la creación y manipulación de procesos.
- Utilizar la biblioteca *os* incluida en el Lenguaje de Programación Python y en especial (no porque sea la única que se va a utilizar) la función *os.fork()* y *os.execlp()*.
- El primer programa consistirá en crear un árbol con nivel de profundidad y factor de ramificación variable.
- El segundo programa consistirá en lanzar una serie de procesos de forma paralela con una estrategia de multiprocesos, similar a la que ejecuta el Sistema Operativo.
- Todo este se pretende hacerlo en un lenguaje diferente y no muy utilizado para este tipo de cursos, el cual es Python.

Documentos a entregar

Esta práctica se tiene que realizar de forma individual:

- Los dos programas desarrollados en Python sin uso de IDE's para desarrollar esta práctica (70 % de la calificación).
- Informe individual de cómo se abordaron cada uno de los problemas planteados (30 % de la calificación), con las mismas especificaciones de las practicas anteriores.

Plazo de entrega

La hora y fecha límite para enviar los programas será el lunes 18 de febrero del 2019 antes de las 10:00 AM. Todos den trabajen de forma individual y enviar su trabajo en el horario establecido para asegurar la revisión de la practica.

Nota. No se recibirá ninguna práctica fuera de ese horario, sin ninguna excepción.

Especificaciones de las partes que conforman la práctica:

Esta práctica se divide en dos partes que describiremos a continuación:

Primera Parte – Generación de Árboles de Procesos con profundidad y factor de ramificación variables:

Para esta primera parte se necesita desarrollar un programa en Python que permita generar una arbolecía de procesos con profundidad y ramificación variable, mediante el uso de la función *os,fork()*.

Además de esta función se utilizarán otras más para lograr un desplegado muy similar al que arroja el comando *ps -fe* (listado de procesos actuales de un usuario), el cual podemos ver en la Figura 1.

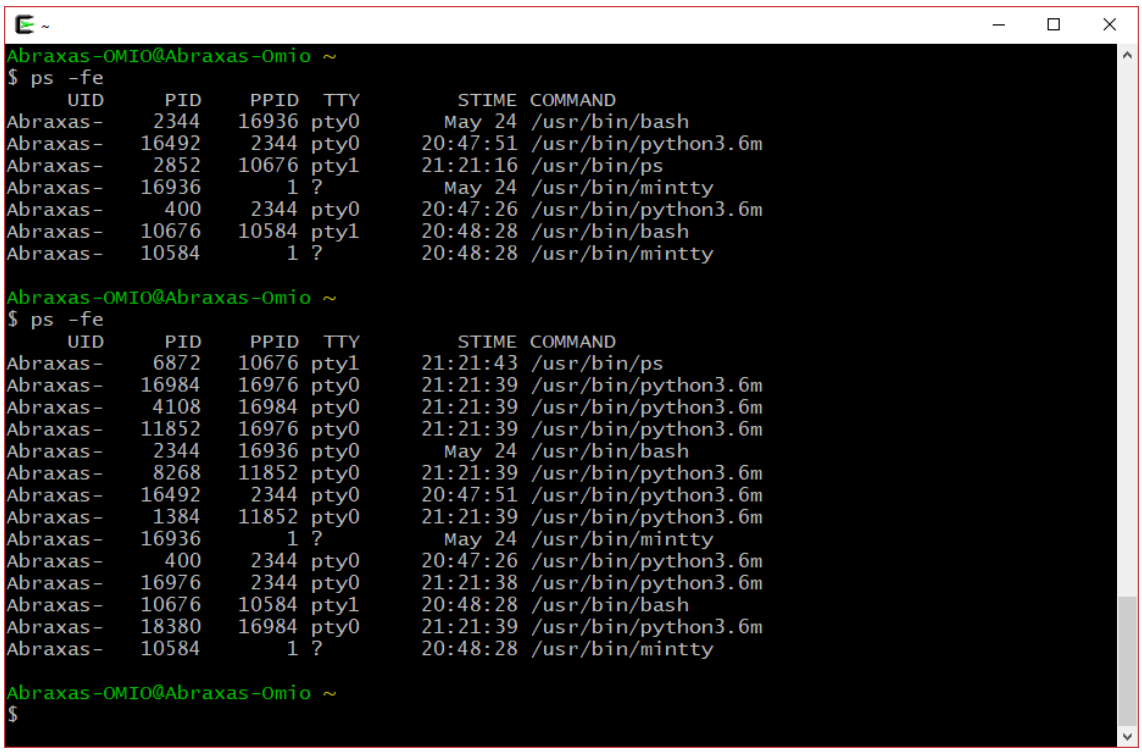


Figura 1. Formato de desplegado del comando *ps -fe*

Como se puede ver en el desplegado aparecen información de los procesos que están siendo creados por un usuario en particular (Abraxas – mi nombre de usuario), el pid del proceso (que se obtiene de la función os.getpid()), el padre de los procesos (que se obtiene de la función os.getppid()), etc. Esta es la información que se solicita en los desplegados de sus respectivos programas.

En resumen, lo que se pide para esta parte es que el programa pida la profundidad y ramificación y en base a estos datos sea capaz de generar los respectivos arboles de procesos. Supóngase que da una profundidad de 2 (las profundidades se cuentan a partir de la siguiente al nodo raíz o proceso padre) y un factor de ramificación de grado 3, en este caso el árbol se visualizaría tal y como se muestra en la Figura 2.

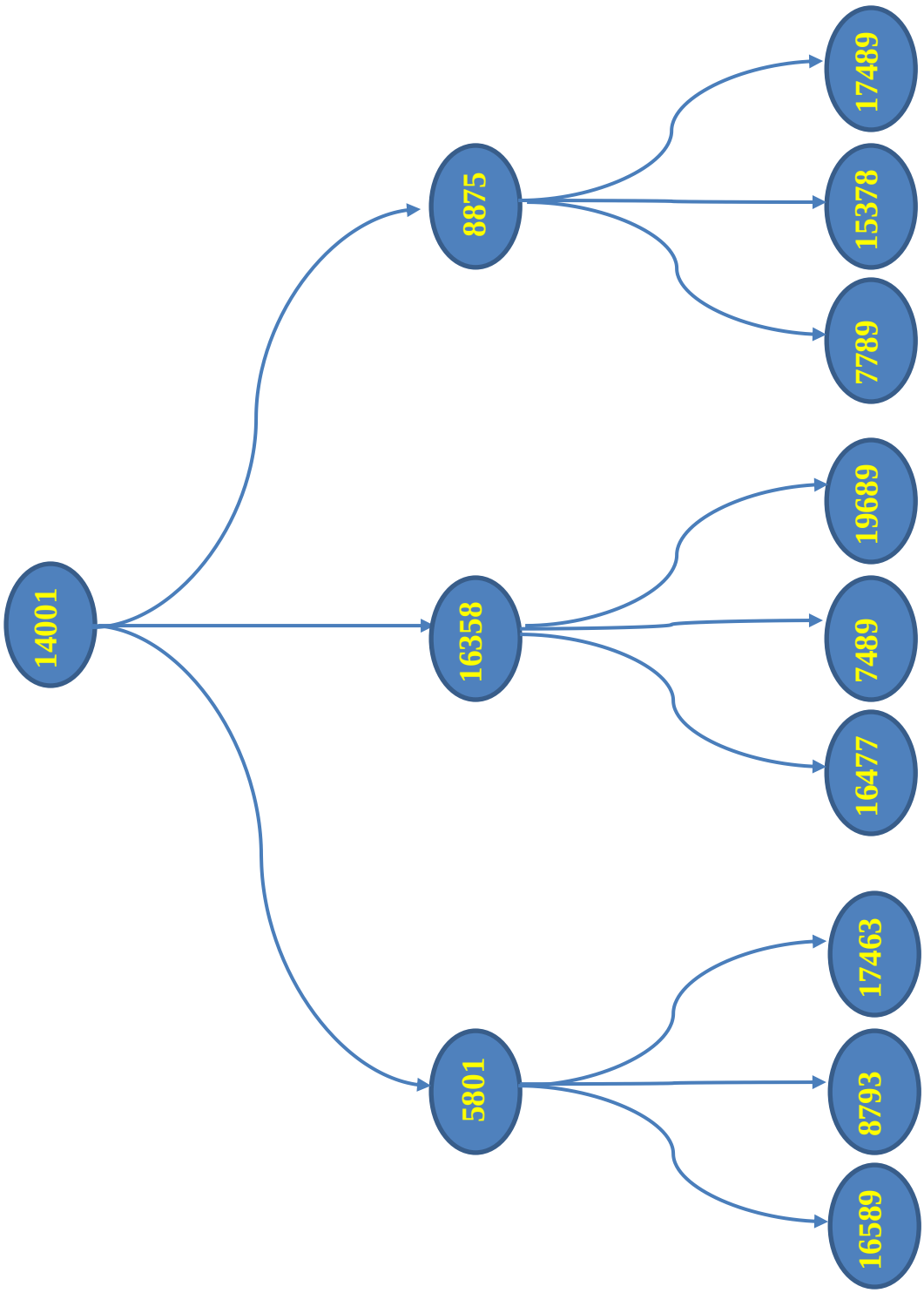
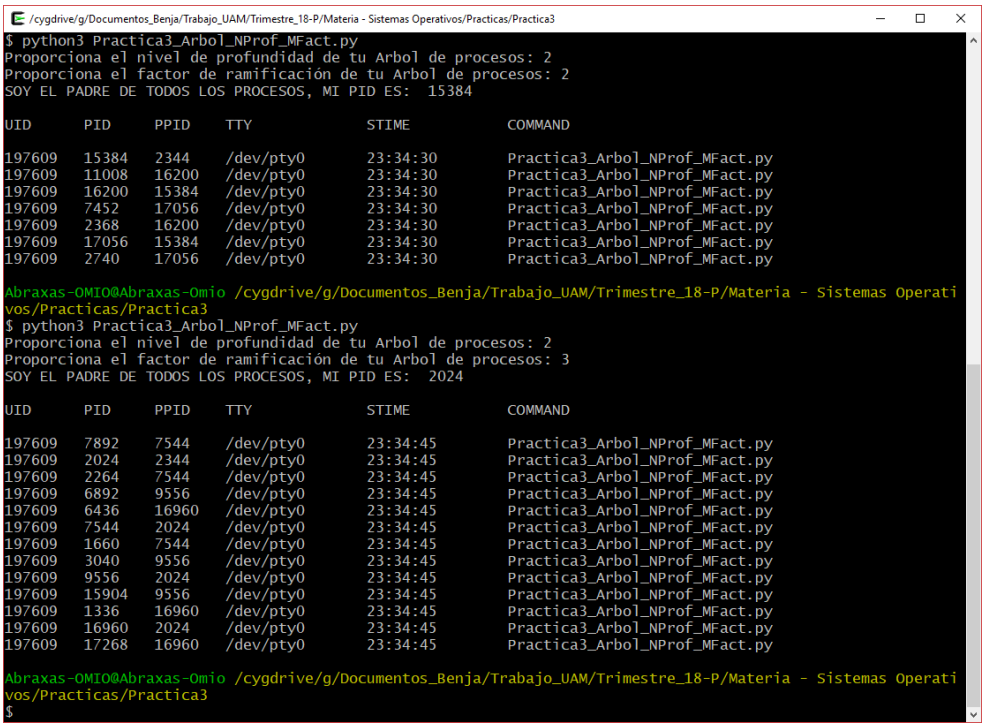
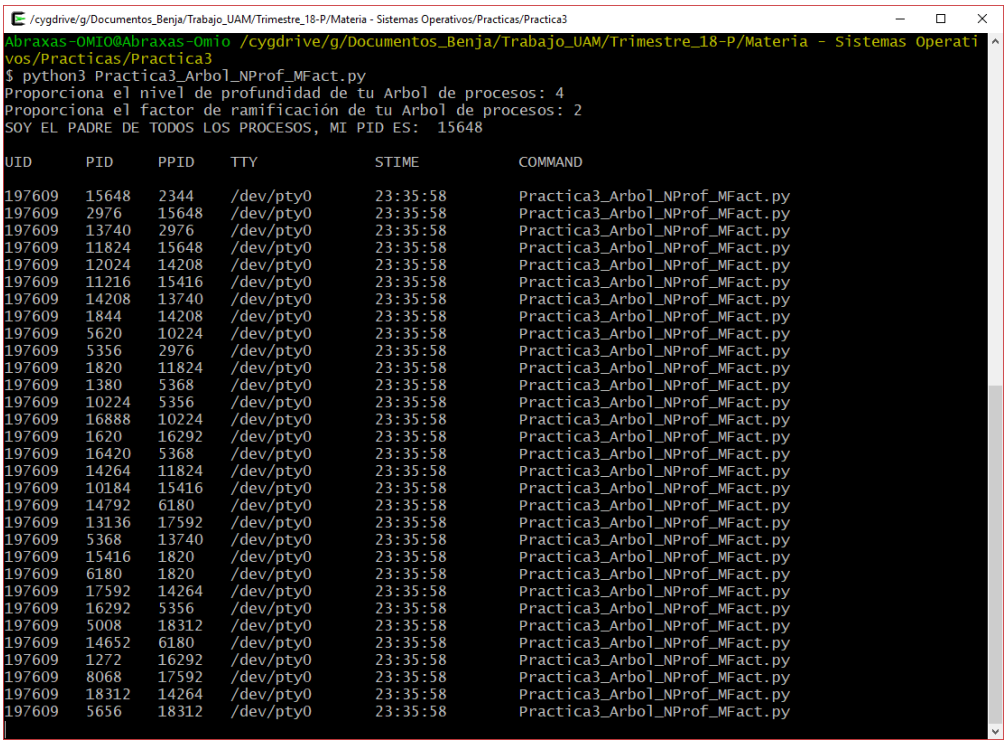


Figura 2. Árbol con profundidad 2 y ramificación 3.

En base a la Figura 2, se puede deducir que el programa podrá formar cualquier tipo de arbolecía y lo mostrará en pantalla como se muestra en la Figura 3 (ambos incisos).



(a)



(b)

Figura 3.a. Profundidad 2, Ramificación 2 y Profundidad 2, Ramificación 3

Figura 3.b. Profundidad 4, Ramificación 2

Segunda Parte: Esquema de MultiProcesos o Pseudoparalelismo:

Para esta parte se necesita implementar una pequeña simulación del como el Sistema Operativo puede generar diferentes procesos, utilizando los aportes del Procesamiento MultiProgramación (dividir la RAM en difrentes pedazos) y Procesamiento MultiTareas o también llamado MultiProcesos o simplemente PseudoParalelismo.

Para esta parte se debe utilizar la función *os.execlp()* junto con la función *os.fork()*. La función *os.execlp()* permite que un proceso generado (forked) tenga una serie de líneas de código a realizarse de forma independiente y paralela al proceso padre que creo este hijo.

Se puede ver que esto es un claro ejemplo de procesos independientes y con su propio código tal y como sucede en el funcionamiento de cualquier sistema operativo. Con los procesadores actuales, no solo un procesador es compartido por todas las tareas o procesos lanzados desde el sistema operativo, el funcionamiento ahora abarca un proceso casi paralelo, al tener diferentes unidades trabajando al mismo tiempo. En el esquema de la Figura 4 se muestra lo que se pretende hacer para esta segunda parte.

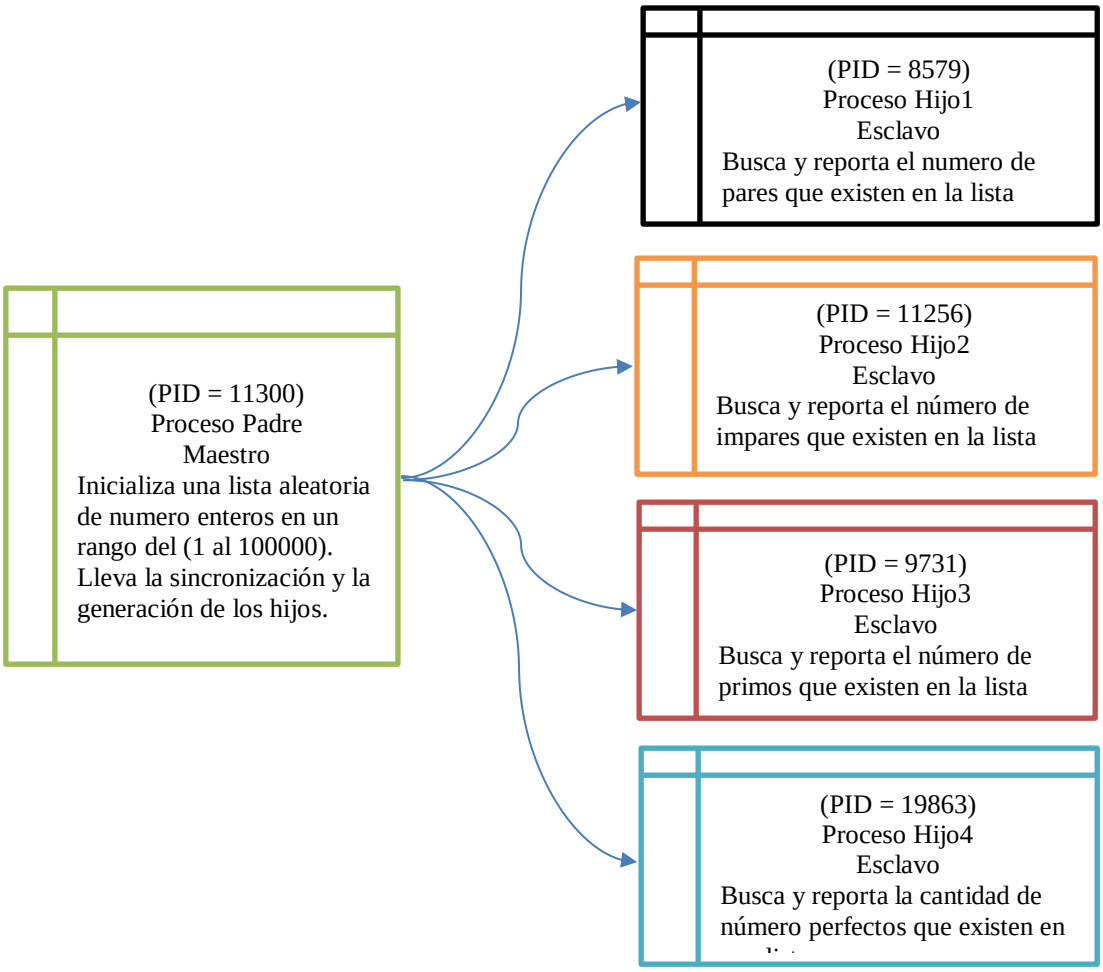


Figura 4. Esquema MultiProcesos para la segunda parte.

Como se puede observar en la Figura 4, el programa arrancara con un proceso denotado con el PID 11300 (esto claramente variara siempre). EL funcionamiento de este proceso será el de maestro, ya que creará a los hijos preparándoles la lista de enteros aleatorios en un rango del [1 al 10000]. Este proceso esperará la finalización de cada hijo creado mediante la función os.wait() y reportara el tiempo de ejecución de la aplicación.

El funcionamiento de los esclavos se puede deducir del diagrama de la Figura 4, pero vamos a detallar el funcionamiento de cada uno, los que si comparten es invocar la función os.execlp() con los parámetros necesarios, tomando en cuenta los ejercicios que se vean en las clases.

1. Hijo1: Este proceso será un esclavo del maestro para buscar los pares que existen en una lista llenada por el padre.
2. Hijo2: Este proceso será un esclavo del maestro para buscar los impares que existen en una lista llenada por el padre.
3. Hijo1: Este proceso será un esclavo del maestro para buscar los primos que existen en una lista llenada por el padre.
4. Hijo1: Este proceso será un esclavo del maestro para buscar los números perfectos que existen en una lista llenada por el padre. Un numero perfecto es un entero positivo en el cual la suma de todos sus divisores sea igual al mismo valor leído, excluyendo el número leído por teclado y al cero. Por ejemplo, el numero 28 sus divisores son 1, 2, 4, 7 y 14, los cuales al ser sumados dan 28.

Al final el proceso maestro reportará el tiempo de ejecución y finalizará la aplicación, tal y como se ve en la Figura 5.

```
Abraxas-OMIO@Abraxas-Omio /cygdrive/g/Documentos_Benja/Trabajo_UAM/Trimestre_1
vos/Practicas/Practica3
$ python3 Practica3_Busqueda.py
Busqueda de numeros impares, pares, primos y perfectos
Soy el Padre de todos los Procesos, mi PID es:  11636
Proporciona el numero de elementos en el arreglo: 1000000
11636 : Inicialiazando la Lista de  1000000  de elementos

        1984 : Método para encontrar los numeros pares en una Lista
        12308 : Método para encontrar los numeros impares en una Lista
        14312 : Método para encontrar los numeros primos en una Lista
        7140 : Método para encontrar los numeros perfectos en una Lista
        1984 : Se encontraron  500022  numeros pares en la lista
        12308 : Se encontraron  499978  numeros impares en la lista
11636 : Mi hijo  (1984, 0)  a finalizado
11636 : Mi hijo  (12308, 0)  a finalizado
        14312 : Se encontraron  123349  numeros primos en la lista
11636 : Mi hijo  (14312, 0)  a finalizado
        7140 : Se encontraron  0  numeros perfectos en la lista
11636 : Mi hijo  (7140, 0)  a finalizado
11636 : El tiempo de ejecucion fue:  542.1428470611572  segundos

Abraxas-OMIO@Abraxas-Omio /cygdrive/g/Documentos_Benja/Trabajo_UAM/Trimestre_1
vos/Practicas/Practica3
$
```

Figura 5. Ejecución de la Segunda Parte de la Practica.