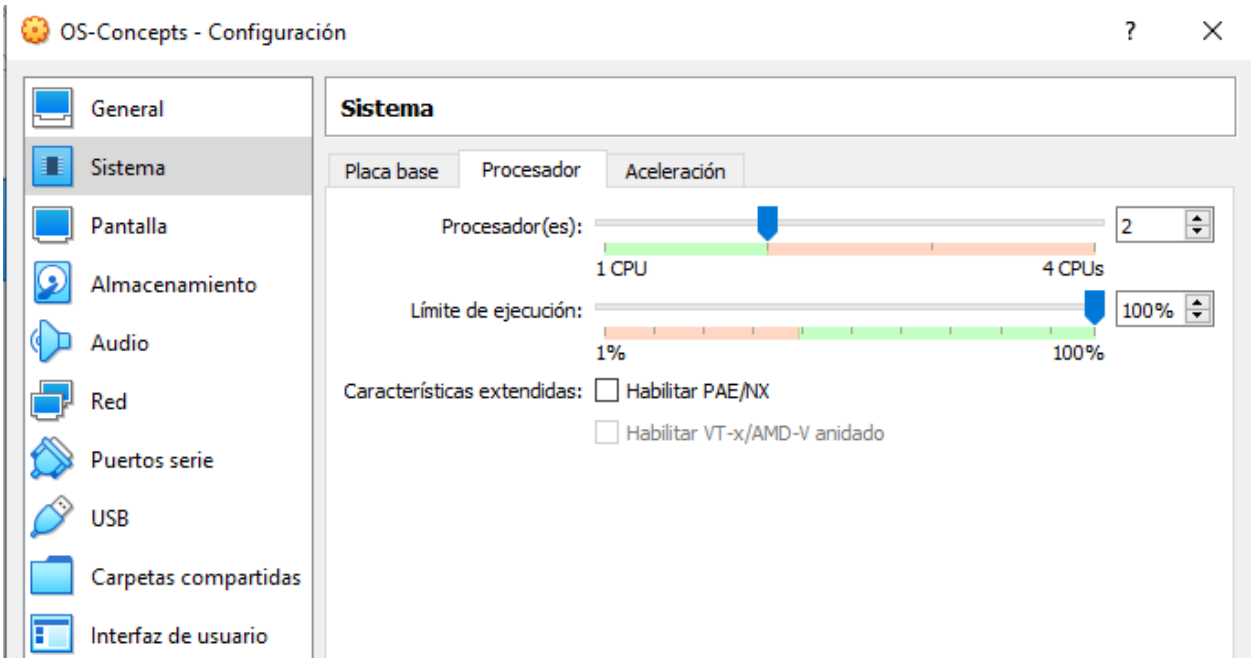
Universidad del Valle de Guatemala
José Gabriel Block Staackmann
carne no. 18935
Sistemas Operativos

HDT#2

Parte 1.

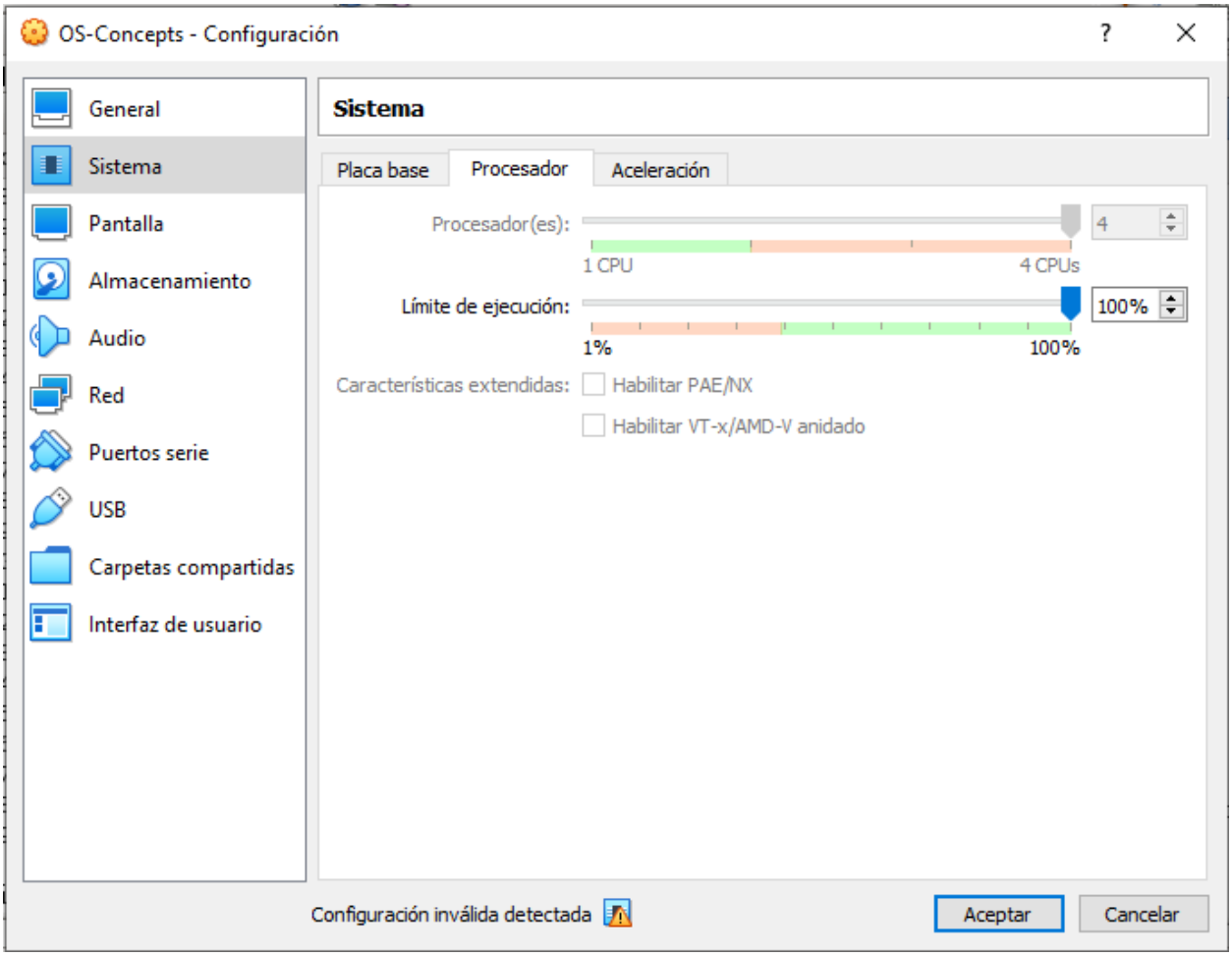



```
os@debian:~$ gcc -o fourforks fourforks.c
os@debian:~$ ./fourforks
forks
forks
os@debian:~$ forks
forks
forks
forks
forks
forks
forks
forks
forks
forks
forks
forks
forks
forks
forks
```

```
forksos@debian:~$ gcc -o forforks forforks.c
os@debian:~$ ./forforks
forks
os@debian:~$
```

- ¿Cuántos procesos se crean en cada uno de los programas?
  En el primero de los 4 forks se crean 16 y en el segundo del for, se crea uno.

- ¿Por qué hay tantos procesos en ambos programas cuando uno tiene cuatro llamadas fork()y el otro sólo tiene una?
  Porque al llamar cada función, hace 4 llamadas diferentes, porque son hechas al mismo tiempo, pero con el for hace la misma llamada 4 veces por eso ambas usan la misma cantidad de procesos.



Parte 2.

```
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ gcc -o clo
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock1
6104.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock1
5971.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock1
5907.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock1
6749.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock1
6140.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock1
6037.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$
```

```
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock2
90.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock2
120.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock2
93.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$ ./clock2
119.000000
blockmann@blockmann-VirtualBox:~/Documents/Hdt2$
```

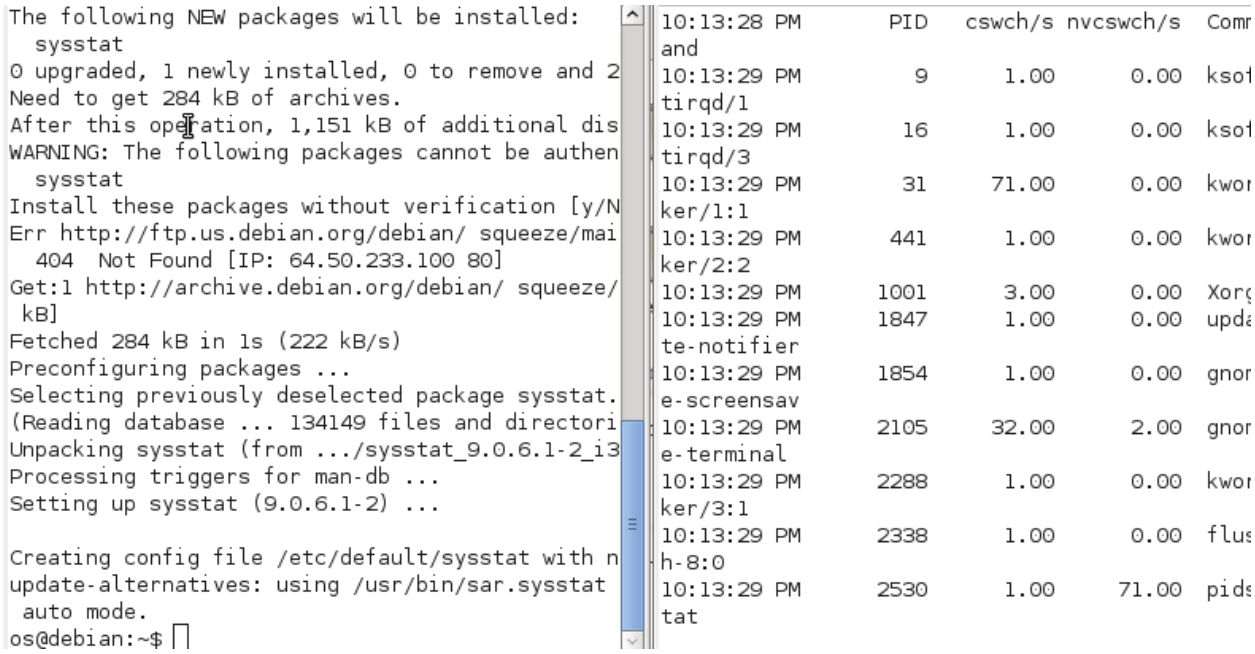
- ¿Cuál, en general, toma tiempos más largos?
  El primer programa tarda mucho más que el segundo.

- ¿Qué causa la diferencia de tiempo, o por qué se tarda más el que se tarda más?
  El orden en el que se organizan y separan en diferentes procesos acelera el tiempo de tardanza de forma notoria.

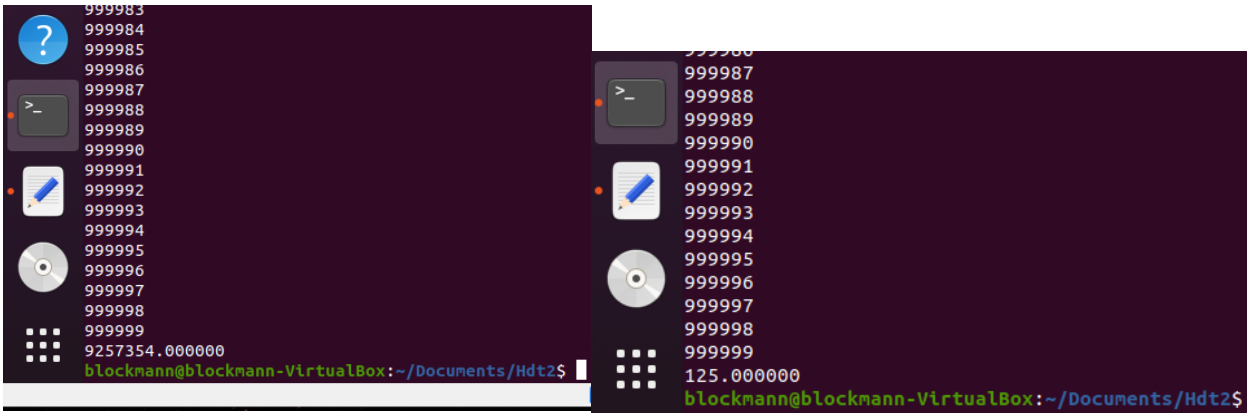
Parte 3.

Tipos de c. de contexto

- Cambio “voluntario”:
  - Proceso realiza llamada al sistema que implica esperar por evento
  - Transición de *en ejecución* a *bloqueado*
  - Ejemplos: leer del terminal o bajar un semáforo cerrado
  - Motivo: Eficiencia en el uso del procesador
- Cambio “involuntario”:
  - S.O. le quita la UCP al proceso
  - Transición de *en ejecución* a *listo*
  - Ejemplos: fin de rodaja de ejecución o pasa a *listo* proceso bloqueado de mayor prioridad
  - Motivo: Reparto del procesador



- ¿Qué tipo de cambios de contexto incrementa notablemente en cada caso, y por qué?
  Al teclear cosas o mover el mouse sobre otras, incrementan los pidstats y los Xorgs al usar un programa aumentaron los números de keyworkers.



- ¿Qué diferencia hay en el número y tipo de cambios de contexto de entre programas?
  Al correr el programa concurrente de forks, se puede notar una disminución de los cambios de contexto involuntarios, y los voluntarios aumentan.

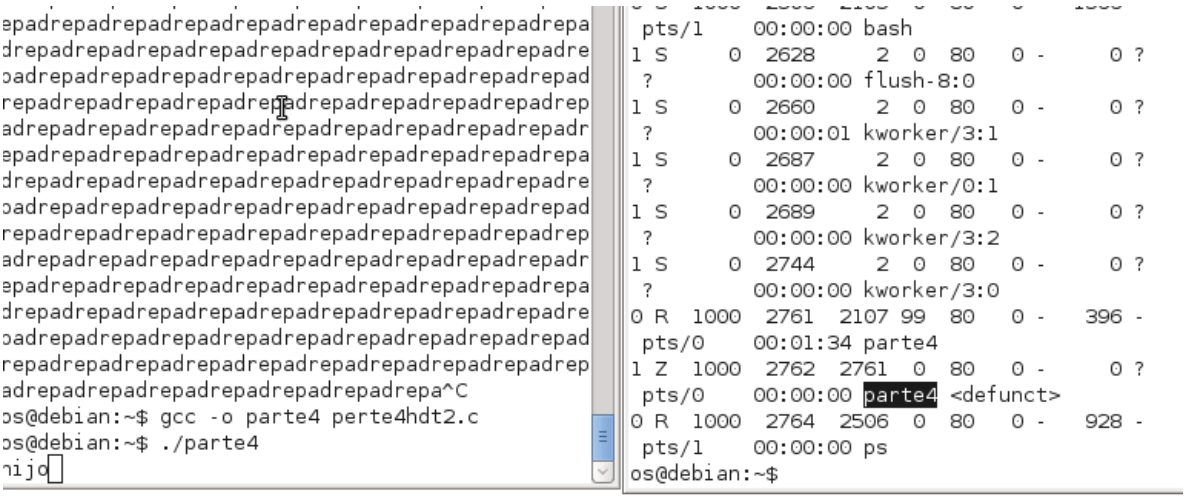
- ¿A qué puede atribuir los cambios de contexto voluntarios realizados por sus programas?
  Es por los forks() y su forma de funcionar de manera secuencial.

- ¿A qué puede atribuir los cambios de contexto involuntarios realizados por sus programas?
  Por los procesos del SO, al llevar un tiempo prolongado de corrida se dan los cambios de contexto involuntarios.

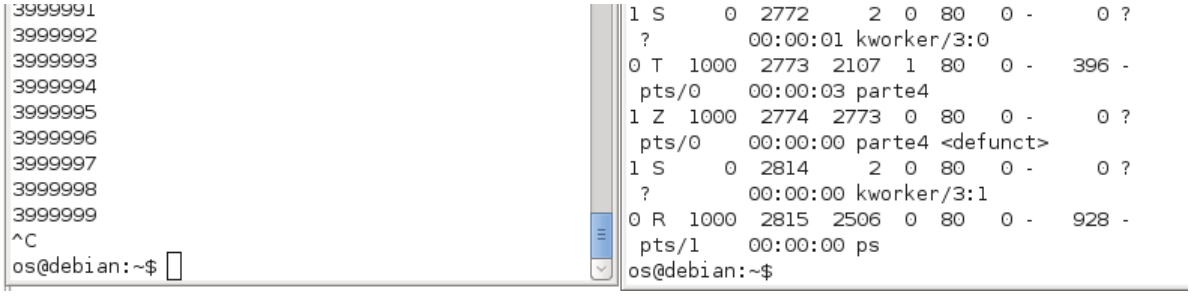
- ¿Por qué el reporte de cambios de contexto para su programa con fork()s muestra cuatro procesos, uno de los cuales reporta cero cambios de contexto?
  Porque un fork() no espera a ser ejecutado, por lo tanto este se realiza rápido y por lo tanto se generan los procesos hijos.

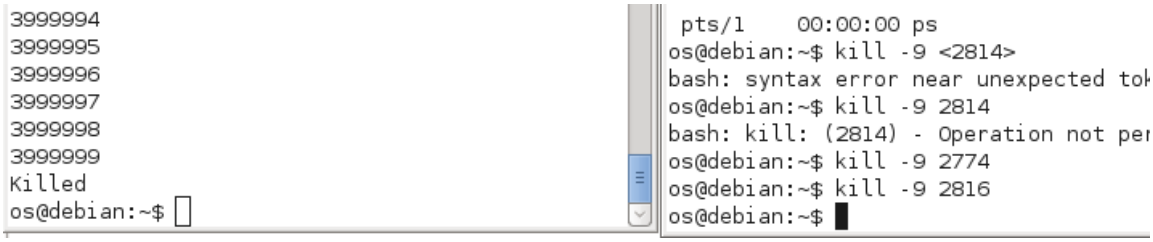
- ¿Qué efecto percibe sobre el número de cambios de contexto de cada tipo?
  Aumentan cambios involuntarios.

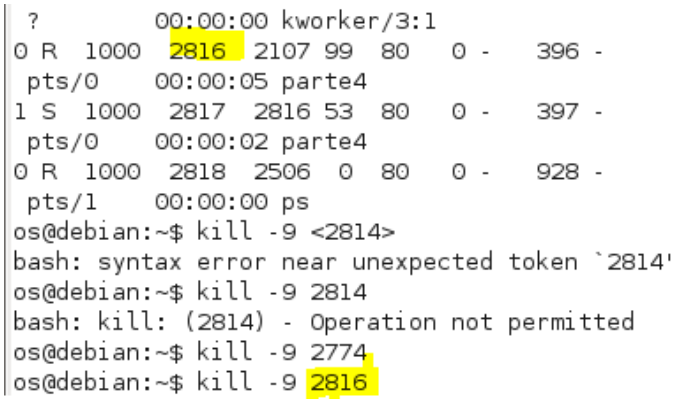
Parte 4.



- ¿Qué significa la Z y a qué se debe?
  Z significa que es un proceso zombie, o sea que este a terminado su tarea, pero el producto de esta no ha sido usado de ninguna forma.



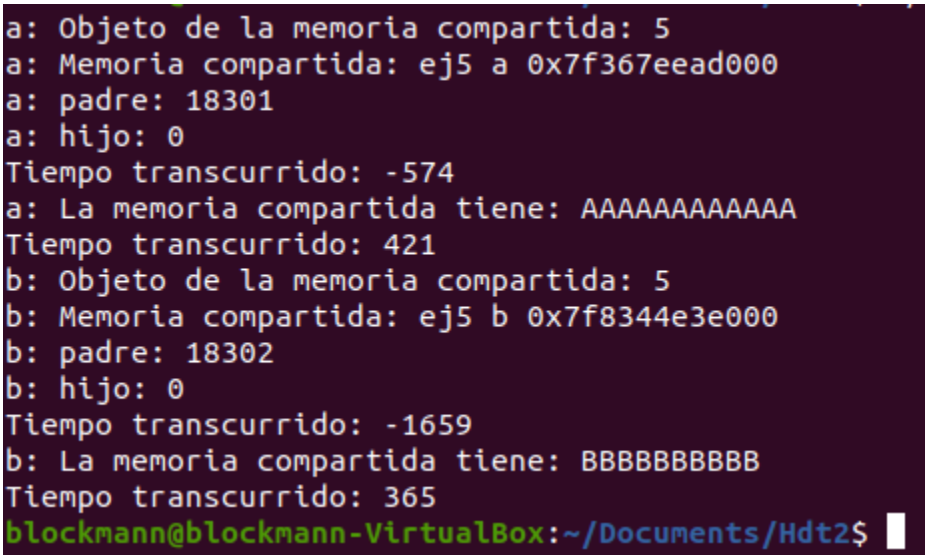




- ¿Qué sucede en la ventana donde ejecutó su programa?
  En la Ventana que se usó el programa, se está contando de 0 a 3999999.

- ¿Quién es el padre del proceso que quedó huérfano?
  El padre es el 2816, y quedo huérfano el 2817.

Parte 5.



- ¿Qué diferencia hay entre realizar comunicación usando memoria compartida en lugar de usando un archivo de texto común y corriente?
  La memoria compartida suele ser más rápida con el paso de mensajes porque usa menos al sistema operativo porque no requiere de un protocolo ni preparación y manejo especial de la información. Por otro lado está la transmisión de mensajes send() y receive() que se hacen a través de un enlace de comunicación y se diferencia entre directa e indirecta.
- ¿Por qué no se debe usar el file descriptor de la memoria compartida producido por otra instancia para realizar el mmap?
  El file descriptor no puede ser utilizado más de una vez por instancias y además el file descriptor es necesario para que el mmap funcione adecuadamente.

- ¿Es posible enviar el output de un programa ejecutado con execl otro proceso por medio de un pipe? Investigue y explique cómo funciona este mecanismo en la terminal (e.g., la ejecución de ls | less).
  La ejecución ls busca el path en las ubicaciones posibles del directorio, una vez encontrado entonces se ejecuta execl y por un pipe se envía el resultado de la búsqueda.

- ¿Cómo puede asegurarse de que ya se ha abierto un espacio de memoria compartida con un nombre determinado? Investigue y explique errno.
  Errno es un int var que se define gracias a las llamadas al sistema que retornan un tipo de error o no. Con errno se utiliza shm_open que sirve para saber si el espacio en la memoria se generó correctamente.

- ¿Qué pasa si se ejecuta shm_unlink cuando hay procesos que todavía están usando la memoria compartida?
  Todos los procesos dejan de hacer el mapeo, o sea que deja de funcionar y por lo tanto se libera el espacio de memoria compartida.

- ¿Cómo puede referirse al contenido de un espacio en memoria al que apunta un puntero? Observe que su programa deberá tener alguna forma de saber hasta dónde ha escrito su otra instancia en la memoria compartida para no escribir sobre ello.
  Se podría con el buffer ya que muestra la información del puntero para poder escribir en la memoria compartida.

- Imagine que una ejecución de su programa sufre un error que termina la ejecución prematuramente, dejando el espacio de memoria compartida abierto y provocando que nuevas ejecuciones se queden esperando el file descriptor del espacio de memoria compartida. ¿Cómo puede liberar el espacio de memoria compartida “manualmente”?
  Con munmap() ya que este elimina un mapeo en la memoria compartida.
  https://pubs.opengroup.org/onlinepubs/009695399/functions/munmap.html

- Observe que el programa que ejecute dos instancias de ipc.c debe cuidar que una instancia no termine mucho antes que la otra para evitar que ambas instancias abran y cierren su propio espacio de memoria compartida. ¿Aproximadamente cuánto tiempo toma la realización de un fork()? Investigue y aplique usleep.
  Toma alrededor de 800 tics de la función clock() microsegundos con buffer de 100 . Usleep detiene la a un proceso por un cierto tiempo determinado.