Sistemas Operativos - Virtualización del CPU

Ejercicio 1. En un sistema operativo que implementa procesos se ejecutan instancias del proceso `pi` que computa los dígitos de π con precisión arbitraria.

```
$ time pi 1000000 > /dev/null & ... & time pi 1000000 > /dev/null &
```

Y se registran los siguientes resultados, donde en las mediciones se muestra $(real, user)$, es decir el tiempo del reloj de la pared (walltime) y el tiempo que insumió de CPU (cputime).

#Instancias	Medición	Descripción
1	(2.56,2.44)	
2	(2.53,2.42), (2.58,2.40)	
1	(3.44,2.41)	
4	(5.12,2.44), (5.13,2.44), (5.17,2.46), (5.18,2.46)	
3	(3.71,2.42), (3.85,2.42), (3.86,2.44)	
2	(5.04,2.36), (5.09,2.43)	
4	(7.67,2.41), (7.67,2.44), (7.73,2.44), (7.75,2.46)	

(a) ¿Cuántos núcleos tiene el sistema?

(b) ¿Porqué a veces el cputime es menor que el walltime?

(c) Indique en la **Descripción** que estaba pasando en cada medición.

a. En la primera sentencia, se observa al menos un core, pero ya en la segunda contamos con dos cores ya que los tiempos de CPU varían entre los procesos. En la tercera sentencia no se nos revela nada nuevo, al menos se está usando un núcleo. Luego, en la 4 sentencia vemos que en dos tiempos diferentes, se están ejecutando dos procesos simultáneos, por lo que hay 2 core al menos. En la siguiente similar, al menos dos núcleos, y análogas las dos últimas sentencias. Finalmente, el sistema cuenta con al menos dos núcleos.
b. Puede ser por muchas causas, las que podrían ser adecuadas en este contexto son:
    - Procesamiento en paralelo.
    - Multitasking.
    - Cambios de contexto.
    - Otros procesos en el sistema.

Todos ellos están presentes en un sistema que tiene varios núcleos.

Ejercicio 2. En un sistema operativo que implementa **procesos** e **hilos** se ejecutan el siguiente proceso. Explique porque ahora $walltime < cputime$.

```
$ time ./dgemm 2000 2000 2000
test!
m=2000,n=2000,k=2000,alpha=1.200000,beta=0.001000,sizeofc=4000000

real  0m1.027s
user  0m1.752s
```

Definamos primero

- "User time" (tiempo de usuario) : se refiere al tiempo total que la CPU ha pasado ejecutando código en el espacio de usuario del proceso.
- "Real time" (tiempo real) : es la cantidad total de tiempo que ha transcurrido en el reloj del sistema desde que comenzó la ejecución del proceso.

En general, cuando el walltime es menor que el CPUtime se debe a que el sistema está utilizando alguna herramienta de optimización a la hora de la ejecución. En este caso, el programa está ejecutándose paralelamente, lo que hace que el tiempo de uso de la CPU sea menor al tiempo transcurrido en la realidad.

Ejercicio 3. Describir donde se cumplen las condiciones *user<real*, *user=real*, *real<user*.

- User < Real : en esta condición, el proceso ha utilizado menos tiempo de CPU del que ha pasado en el reloj del sistema. Esto podría deberse a que el proceso ha estado esperando recursos, realizando operaciones de I/O o ha sido interrumpido por otras tareas.
- User = Real : el tiempo que utilizó el procesador para ejecutar el proceso es el mismo que el de reloj, lo que quiere decir es que el proceso se ejecutó sin pausas ni interrupciones .
- Real < User : el proceso aprovechó el paralelismo de los múltiples núcleos del CPU, lo que genera la realización de más trabajo en la chip, en un periodo de tiempo menor.

Ejercicio 4. Un programa define la variable `int x=100` detro de `main()` y hace `fork()`.

(a) ¿Cuánto vale `x` en el proceso hijo?

(b) ¿Qué le pasa a la variable cuando el proceso padre y el proceso hijo le cambian de valor?

(c) Contestar nuevamente las preguntas si el compilador genera código de máquina colocando esta variable en un registro del microprocesador.

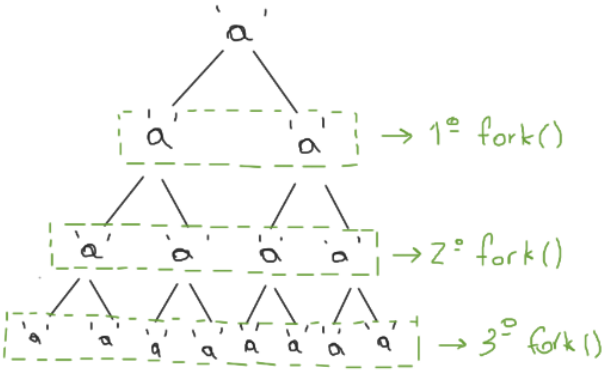
a. Vale lo mismo, ya que el proceso hijo y padre son totalmente iguales pero independientes entre sí.
b. Después de que se crea el proceso hijo, cada proceso tiene su propia copia de la variable x. Si tanto el proceso padre como el proceso hijo cambian el valor de x, estos cambios sólo afectarán a su propia copia de la variable y no tendrán ningún efecto en el otro proceso. Por lo tanto, los cambios en x en el proceso padre no afectarán el valor de x en el proceso hijo y viceversa.
c. En este caso, dado que un proceso hijo es una copia exacta del espacio de direcciones del proceso padre, si la variable x se coloca en un registro del procesador, el proceso hijo también tendrá su propia copia de la variable en un registro separado.

Ejercicio 5. Indique cuantas letras "a" imprime este programa, describiendo su funcionamiento.

```
printf("a\n");
fork();
printf("a\n");
fork();
printf("a\n");
fork();
printf("a\n");
```

Generalice a n `forks`. Analice para n=1, luego para n=2, etc., busque la serie y deduzca la expresión general en función del n.

Hagamos la traza de ejecución, y veamos cuántas 'a' se llegan a imprimir.



Se imprimen 15 ‘a’. Veamos para ‘n’ forks.

- n = 1 → Se imprime una a.
- n = 2 → Se imprimen tres a.
- n = 3 → Se imprimen 15 a.

Podemos ver que sigue la forma de la ecuación 2^n - 1.

Ejercicio 6. Indique cuantas letras “a” imprime este programa

```
char * const args[] = {"/bin/date", "-R", NULL};
execv(args[0], args);
printf("a\n");
```

A lo sumo, se imprime una sola ‘a’. Pero para que eso pase, el execv no se debería ejecutar correctamente, dado que al ejecutarse exitosamente reemplazaria el proceso actual por “/bin/date”.

Ejercicio 7. Indique que hacen estos programas.

```
int main(int argc, char ** argv) {
        if (0<--argc) {
                argv[argc] = NULL;
                execvp(argv[0], argv);
        }

        return 0;
}
```

```
int main(int argc, char ** argv) {
        if (argc<=1)
                return 0;
        int rc = fork();
        if (rc<0)
                return -1;
        else if (0==rc)
                return 0;
        else {
                argv[argc-1] = NULL;
                execvp(argv[0], argv);
        }
}
```

Básicamente, este código está diseñado para tomar argumentos de la línea de comandos y ejecutar un programa especificado por esos argumentos.

1. int main(int argc, char ** argv): Esta es la definición de la función principal main, que toma dos parámetros. argc es el número de argumentos pasados en la línea de comandos (incluyendo el nombre del programa), y argv es un arreglo de cadenas que contiene los argumentos.

2. if (0<--argc) : Lo que parece estar intentando hacer es verificar si hay al menos un argumento además del nombre del programa (porque se está comparando con cero), pero la sintaxis está incorrecta.

3. argv[argc] = NULL; : Aquí se está configurando el último elemento del arreglo argv para que sea un puntero nulo. Esto es una práctica común en C para indicar el final de la lista de argumentos en funciones relacionadas con la ejecución de comandos.

4. execvp(argv[0], argv);: Esta línea intenta ejecutar un programa especificado por argv[0] con los argumentos contenidos en el arreglo argv. La función execvp busca el programa en las rutas de búsqueda del sistema y lo ejecuta. Si esta llamada tiene éxito, el proceso actual se reemplazará por el nuevo programa y el código restante después de esta línea no se ejecutará.

El otro código, intenta ejecutar un nuevo programa utilizando execvp() con los argumentos proporcionados en la línea de comandos, después de haber creado un proceso hijo usando fork(). El enfoque es permitir que el programa se ejecute en segundo plano utilizando el proceso hijo mientras que el proceso padre podría ser liberado.

1. if (argc <= 1) return 0;: Esta línea verifica si no se han proporcionado argumentos adicionales en la línea de comandos (aparte del nombre del programa). Si no hay argumentos adicionales, el programa simplemente finaliza con un código de retorno 0. Esto se hace para evitar ejecutar fork() innecesariamente.

2. int rc = fork();: Aquí se llama a la función fork(), que crea un nuevo proceso hijo duplicando el proceso actual. La variable rc contendrá el valor de retorno de fork(). Si rc es negativo, significa que ha ocurrido un error en la creación del proceso hijo. Si rc es igual a 0, eso significa que el proceso actual es el proceso hijo. Si rc es mayor que 0, entonces el proceso actual es el proceso padre.

3. if (rc < 0) return -1;: Esta línea verifica si fork() ha devuelto un valor negativo, lo que indica un error en la creación del proceso hijo. En tal caso, el programa finaliza con un código de retorno -1.

4. else if (0 == rc) return 0;: Esta línea verifica si rc es igual a 0, lo que significa que el proceso actual es el proceso hijo. Si es así, el proceso hijo simplemente finaliza con un código de retorno 0. Esto evita que el proceso hijo continúe ejecutando el código del padre.

5. else { argv[argc-1] = NULL; execvp(argv[0], argv); }: Si el valor de rc no es negativo ni igual a 0, entonces el proceso actual es el proceso padre. Aquí se configura el último elemento del arreglo argv en NULL, para indicar el final de la lista de argumentos. Luego, se intenta ejecutar el programa especificado por argv[0] con los argumentos proporcionados en argv utilizando execvp(). Si esta llamada tiene éxito, el proceso actual se reemplazará por el nuevo programa y el código restante después de esta línea no se ejecutará.

Ejercicio 8. Si estos programas hacen lo mismo. ¿Para que está la *syscall* dup()? ¿UNIX tiene un mal diseño de su API?

```
close(STDOUT_FILENO);
open("salida.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
printf("¡Mirá mamá salgo por un archivo!");
```

```
fd = open("salida.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
close(STDOUT_FILENO);
dup(fd);
printf("¡Mirá mamá salgo por un archivo!");
```

- La syscall dup() es una parte importante de esta diferencia y tiene un propósito específico en el contexto de la manipulación de descriptores de archivo en UNIX.
- Ambos programas redirigen la salida estándar (STDOUT) hacia el archivo "salida.txt", de modo que el resultado de printf se escribirá en el archivo en lugar de mostrarse en la consola. Sin embargo, el enfoque para lograr esto es diferente.
- En el Programa 1, primero cierras STDOUT con close(STDOUT_FILENO), luego abres el archivo "salida.txt" y, finalmente, usas printf. Esto funcionará, pero cerrar STDOUT puede tener efectos secundarios inesperados si se usa en un programa más grande, ya que STDOUT es un descriptor de archivo importante para la salida estándar del programa.
- En el Programa 2, primero abres el archivo "salida.txt", luego cierras STDOUT y finalmente usas dup(fd) para duplicar el descriptor de archivo abierto (el que apunta al archivo "salida.txt") en el descriptor de archivo STDOUT (que estaba cerrado previamente). Esto es una forma más segura de redirigir la salida estándar sin cerrar STDOUT, lo que evita problemas potenciales con bibliotecas que pueden depender de la salida estándar.

UNIX no es que tenga un mal diseño, sino que ofrece múltiples formas de lograr una tarea para adaptarse a diferentes situaciones y necesidades. La syscall dup() es una parte fundamental de la flexibilidad de UNIX en la manipulación de descriptores de archivo y redirección de la entrada/salida estándar, y su uso en el Programa 2 es más preferible en términos de mantener la robustez y compatibilidad de tu programa con otros componentes del sistema.

Ejercicio 9. Este programa se llama **bomba fork**. ¿Cómo funciona? ¿Es posible mitigar sus efectos?

```
while(1)
        fork();
```

El programa entra en un bucle infinito, y por cada llamada a fork() se duplicarán los procesos en ejecución. Dado que el bucle es infinito, el número de procesos se duplica exponencialmente en cada iteración. Esto significa que después de un tiempo muy corto, se crearán miles de procesos, lo que puede abrumar el sistema operativo y hacer que el sistema sea inoperable.

Ejercicio 10. Para el diagrama de transición de estados de un proceso (OSTEP Figura 4.2), describa cada uno de los 4 (cuatro) escenarios posibles acerca de como funciona (o no) el Sistema Operativo si se quita solo una de las cuatro flechas.

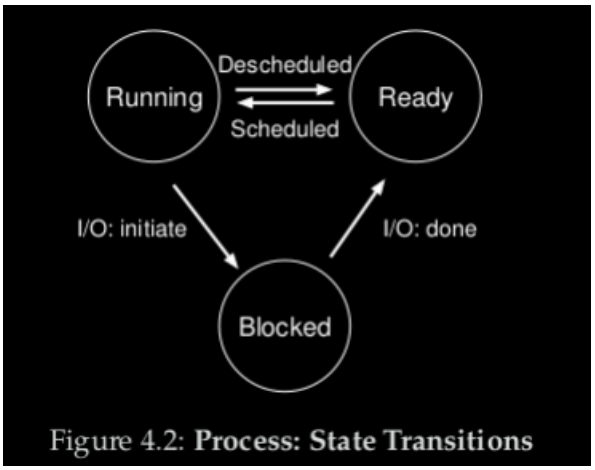


Figure 4.2: **Process: State Transitions**

- Si se quita Descheduled, el kernel solo podría recuperar el control en una I/O request, por lo cuál un proceso podría apoderarse del procesador.
- Si se quita Scheduled el sistema no podría funcionar, ya que no se podría poner a correr un programa en ningún momento.
- Si se quita I/O: initiate el sistema podría funcionar dependiendo del SO. Pero los programas no podrían hacer I/O si no que solo podrían calcular su resultado. Los programas podrían hacer I/O, pero no se bloquearian, y por ende consumirían recursos mientras esperan que el I/O se haga.
- Si se saca I/O: done, el escenario sería similar al anterior solo que los procesos que hagan I/O request no volverían a correr más.

Ejercicio 12. Verdadero o falso. Explique.

(a) Es posible que $user+sys < real$.

(b) Dos procesos no pueden usar la misma dirección de memoria virtual.

(c) Para guardar el estado del proceso es necesario salvar el valor de todos los registros del microprocesador.

(d) Un proceso puede ejecutar cualquier instrucción de la ISA.

(e) Puede haber traps por timer sin que esto implique cambiar de contexto.

(f) `fork()` devuelve 0 para el hijo, porque ningún proceso tiene PID 0.

(g) Las syscall `fork()` y `execv()` están separadas para poder redireccionar los descriptores de archivo.

(h) Si un proceso padre llama a `exit()` el proceso hijo termina su ejecución de manera inmediata.

(i) Es posible pasar información de padre a hijo a traves de `argv`, pero el hijo no puede comunicar información al padre ya que son espacios de memoria independientes.

(j) Nunca se ejecuta el código que está después de `execv()`.

(k) Un proceso hijo que termina, no se puede liberar de la Tabla de Procesos hasta que el padre no haya leído el *exit status* via `wait()`.

a. Verdadero. Cuando el tiempo del sistema (sys) y el tiempo de usuario (user) consumidos por un proceso es menor que el tiempo real (real), significa que el proceso no ha utilizado completamente la capacidad de procesamiento de la CPU durante todo su tiempo de ejecución. Se puede dar por esperas de I/O, multitarea o concurrencia o por la planificación del SO.
   Nota : Sys (Sistema): Representa el tiempo de CPU utilizado por un proceso para ejecutar llamadas al sistema operativo, también conocidas como "syscalls".
b. Falso. Las direcciones de memoria virtual luego son convertidas a direcciones de memoria físicas, por lo que los distintos procesos pueden usar las mismas.
c. Verdadero. Para guardar el estado de un proceso en un sistema informático, generalmente es necesario salvar el valor de los registros del microprocesador. Los registros son ubicaciones de almacenamiento en la CPU que almacenan datos temporales y se utilizan para realizar operaciones y realizar un seguimiento del estado del proceso. Guardar el valor de los registros es esencial para que un proceso pueda reanudarse desde el punto donde se detuvo correctamente.
d. Falso. Hay instrucciones que solo se pueden ejecutar en modo kernel, como las instrucciones para setear el tiempo entre interrupciones.
e. Verdadero. La relación entre las interrupciones generadas por temporizadores (traps de temporizador) y los cambios de contexto depende de la política de planificación de procesos del sistema operativo y de la necesidad de cambiar de proceso en ese momento específico. Puede haber casos en los que se manejen interrupciones de temporizador sin cambiar de contexto y otros casos en los que un cambio de contexto sea necesario.
   Nota :
   - Los traps son interrupciones generadas por el software o el sistema operativo para solicitar servicios específicos del kernel o para manejar eventos excepcionales.
   - El cambio de contexto es el proceso en el que el sistema operativo guarda el estado actual de un proceso en ejecución, incluyendo registros, contador de programa (PC) y otros datos relevantes, y luego carga el estado de otro proceso en la CPU para su ejecución.
f. Ambigua. Es cierto que ningún proceso tiene PID 0, pero eso no obliga a que 0 tenga que ser devuelto al hijo. Al hijo se le asigna 0 para saber que es el hijo, pero podría ser cualquier otro número negativo predefinido, ya que ningún proceso tiene PID negativo.
g. Verdadero. La separación de las llamadas al sistema fork() y execv() es fundamental para permitir la manipulación y redirección eficiente de descriptores de archivo en un proceso antes de ejecutar un nuevo programa con un contexto limpio y definido por execv().
h. Falso. Cuando un proceso padre llama a la función exit(), no termina inmediatamente la ejecución de su proceso hijo. La función exit() se utiliza para finalizar la ejecución del proceso actual (el proceso padre en este caso) y realizar algunas acciones de limpieza, como cerrar descriptores de archivo, liberar memoria, notificar al sistema operativo sobre su finalización, entre otras.
i. Falso. El padre se entera del exitcode del hijo, y es justamente este último quien le pasa esa información.

j. Verdadero.

Ejercicio 13. Dados tres procesos CPU-bound puros A, B, C con $T_{arrival}$ en 0 para todos y T_{cpu} de 30, 20 y 10 respectivamente. Dibujar la línea de tiempo para las políticas de planificación FCFS y SJF. Calcular el promedio de $T_{turnaround}$ y $T_{response}$ para cada política.

- T_Arrival A = 0 y T_Cpu A = 30
- T_Arrival B = 0 y T_Cpu B = 20
- T_Arrival C = 0 y T_Cpu C= 10

- T_turnaround = T_completion - T_arrival
- T_response = T_firstrun - T_arrival

Veamos la línea de tiempo en políticas FCFS :

Ready	B - C	B - C	B - C	B - C	C	C	
CPU	A	A	A	A	B	B	C
Time	0	10	20	30	40	50	60

- T_turnaround A = 30 - 0 → 30 - T_response A = 0 - 0 → 0
- T_turnaround B = 50 - 0 → 50 - T_response B = 40 - 0 → 40
- T_turnaround C = 60 - 0 → 60 - T_response C = 60 - 0 → 60

- Promedio T_turnaround = (30 + 50 + 60) / 3 → 46.666…
- Promedio T_response = (0 + 40 + 60) / 3 → 33.333…

Por otro lado, la línea de tiempo en SJF, se veria asi :

Ready	A - B	A - B	A	A			
CPU	C	C	B	B	A	A	A
Time	0	10	20	30	40	50	60

- T_turnaround A = 60 - 0 → 60 - T_response A = 40 - 0 → 40
- T_turnaround B = 30 - 0 → 50 - T_response B = 30 - 0 → 30
- T_turnaround C = 10 - 0 → 10 - T_response C = 0 - 0 → 0

- Promedio T_turnaround = (60 + 50 + 10) / 3 → 40
- Promedio T_response = (40 + 30 + 0) / 3 → 23.333…

Ejercicio 14. Para esto procesos CPU-bound puros dibujar la línea de tiempo y completar la tabla para las políticas apropiativas (con flecha de running a ready): STCF, RR(Q=2). Calcular el promedio de $T_{turnaround}$ y $T_{response}$ en cada caso.

Proceso	$T_{arrival}$	T_{CPU}	$T_{firstrun}$	$T_{completion}$	$T_{turnaround}$	$T_{response}$
A	2	4				
B	0	3				
C	4	1				

La línea de tiempo, con respecto a la planificación STCF, seria de la forma :

Ready			A	A	C	A			
CPU	B	B	B	B	A	C	A	A	A
Time	0	1	2	3	4	5	6	7	8

Mientras que con RR y Q = 2 :

Ready			A	B	B - C	A - C	C	C	
CPU	B	B	B	A	A	B	A	A	C
Time	0	1	2	3	4	5	6	7	8

Ejercicio 15. Las políticas de planificación se pueden clasificar en dos grades grupos: por lotes (batch) e interactivas. Otra criterio posible es si la planificación necesita el T_{CPU} o no. Clasificar FCFS, SJF, STCF, RR, MLFQ según estos dos criterios.

Política FCFS (First-Come, First-Served):
- Clasificación: Batch.
- Razón: La política FCFS no prioriza la interacción en tiempo real y asigna recursos de procesador en el orden en que llegan los procesos. Esto la hace más adecuada para aplicaciones de procesamiento por lotes que no requieren interacción inmediata con los usuarios.
- No necesita el tiempo de CPU.

Política SJF (Shortest Job First):
- Clasificación: Batch.
- Razón: SJF asigna recursos a los procesos en función de la longitud de sus tareas, sin tener en cuenta la interacción en tiempo real. Es eficiente en términos de tiempos de ejecución, pero no se adapta bien a aplicaciones interactivas.
- Necesita el tiempo de CPU.

Política STCF (Shortest Time-to-Completion First):
- Clasificación: Interactive.
- Razón: STCF prioriza los procesos que tienen el menor tiempo estimado de finalización, lo que puede proporcionar una respuesta más rápida a las solicitudes de los usuarios. Es especialmente útil en entornos interactivos donde la latencia es crítica.
- Necesita CPU time.

Política RR (Round Robin):
- Clasificación: Interactive.
- Razón: RR es una política de planificación de tiempo compartido que asigna una cantidad fija de tiempo de CPU a cada proceso en un ciclo de turnos. Esto permite una respuesta más equitativa y predecible para los procesos interactivos y es adecuado para entornos donde la interacción en tiempo real es importante.
- No necesita CPU time.

Política MLFQ (Multilevel Feedback Queue):
- Clasificación: Puede ser utilizada para ambos, Batch e Interactive.
- Razón: MLFQ es una política flexible que se puede configurar para adaptarse a diferentes tipos de aplicaciones. Puede ser adecuada para entornos batch al ajustar las prioridades de las colas según la carga de trabajo, y también es adecuada para aplicaciones interactivas al asignar prioridades en función del comportamiento del proceso.
- No necesita CPU time.

Ejercicio 16. Considere los siguientes procesos que mezclan ráfagas de CPU con ráfagas de IO.

Proceso	$T_{arrival}$	T_{CPU}	T_{IO}	T_{CPU}	T_{IO}	T_{CPU}	T_{IO}	T_{CPU}
A	0	3	5	2	4	1		
B	2	8	1	6				
C	1	1	3	2	5	1	4	2

Realice el diagrama de planificación para un planificador RR (Q=2). Marque bien cuando el proceso está bloqueado esperando por IO.

Block				C	C	C A	A	A	A	A C	C	C	C	C A	A	A	A C	C	C B	C							
Ready		C	C B	A B	B			B	B			B	B		C	B					C	B	B				
RUNNING	A	A	A	C	A	B	B	C	C	B	B	A	A	B	B	C	B	B	A	B	B	C	C	B	B	B	B
Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	16

Ejercicio 18. Verdadero o falso. Explique.

(a) Cuando el planificador es apropiativo (con flecha de Running a Ready) no se puede devolver el control hasta que no pase el quantum.

(b) Entre las políticas por lote FCFS y SJF, hay una que siempre es mejor que la otra respecto a $T_{turnaround}$.

(c) La política RR con *quanto* $= \infty$ es FCFS.

(d) MLFQ sin *priority boost* hace que algunos procesos puedan sufrir de *starvation* (inanición).

(e) En MLFQ acumular el tiempo de CPU independientemente del movimiento entre colas evita hacer trampas como `yield()` un poquitito antes del quantum.

a. Falso, si el proceso termina antes del Quantum se devuelve el control.
b. Falso, en el caso de que los tiempos sean distintos y decrecientes - crecientes.
c. Verdadero, ya que se ejecutaría un proceso hasta que termine, y luego el siguiente.
d. Verdadero, si aparecen varios procesos de tiempo de CPU reducido, entonces los procesos en la cola de prioridad baja nunca tendrían tiempo de procesado.
e. Verdadero, a pesar de que existen otras formas.