Mecanismos

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.

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
```

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

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.

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.

Ejercicio 6. Indique cuantas letras "a" imprime este programa

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

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);
        }
}
```

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!");
```

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

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

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.

Ejercicio 11. Dentro de `xv6` el archivo `x86.h` contiene `struct trapframe` donde se guarda toda la información cuando se produce un trap. Indicar que parte es la que apila el hardware cuando se produce un trap y que parte apila el software.

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()`.

Políticas

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.

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				

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.

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.

Ejercicio 17. Realice el diagrama de planificación para un planificador MLFQ con cuatro colas (Q=1, 2, 4 y 8) para los siguientes procesos CPU-bound:

Proceso	$T_{arrival}$	T_{CPU}
A	0	7
B	1	3
C	2	4
D	4	3
E	7	4

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.