## Mecanismos

Ejercicio 1. En un sistema operativo que implementa procesos se ejecutan instancias del proceso `pi` que computa los dígitos de $\pi$ 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.

**Línea 1**: Es un proceso que está corriendo en una sola CPU. Fijate que Walltime>CPUTime porque hay pérdida de tiempo en cambios de contexto y de cómputo dentro del kernel.
**Línea 2**:Son dos procesos corriendo cada uno en su core o CPU y esto muestra que al menos hay dos cores. Notar que lanzar 2 procesos tarda lo mismo que uno solo, luego, hay DOS unidades de cómputo
**Línea 3**: Proceso que está corriendo en una sola CPU. Walltime > CPUTime presumiblemente por procesos previamente activos, los cuales hacen que el walltime sea mayor que en la línea 1, pero no así el CPUTime.
**Línea 4**: En un mismo CPU se corren 4 procesos (simultáneamente) que tardan lo mismo (2 cores). Esto refuerza la hipótesis de la línea 2, que tenemos dos núcleos.
**Línea 5**: Se corren 3 procesos, un núcleo A toma 2 y el otro núcleo B toma 1. Una vez el núcleo B acaba con su proceso, ¿ayuda al núcleo A? O se van turnando entre sí cada proceso? Rta: Se distribuye la carga de los 3 procesos entre los 2 núcleos para que ningún núcleo quede inactivo (fairness). Se busca que todos los procesos terminen a la vez.
**Línea 6**: Se puede asumir que en cada core corre un proceso pi que compiten con otros procesos que alargan el walltime.
**Línea 7**: Asumimos que empiezan a trabajar un proceso en cada núcleo, todos ellos compiten con otros procesos y tarda más en terminar lo cual genera que (por ej) el último tarde 7.75. Procesos previos agregan carga de procesado y por lo tanto aumentan walltime.

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 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
```

Primero, ¿Qué es un hilo? Un hilo en una CPU es una unidad de ejecución que forma parte de un proceso y permite realizar múltiples tareas en paralelo dentro de ese proceso, ayudando a mejorar la eficiencia y el rendimiento del sistema.

Un hilo es una característica más relacionada con los **procesos** que con la CPU en sí misma. Aquí está una forma de verlo:

1. **Hilo y Proceso**: Un hilo es una parte de un proceso. Dentro de un proceso, puedes tener varios hilos, y todos estos hilos comparten el mismo espacio de memoria y recursos del proceso, como las variables globales y los archivos abiertos. Por ejemplo, un navegador web puede tener múltiples hilos para manejar diferentes pestañas o tareas en paralelo (como cargar contenido en segundo plano mientras el usuario interactúa con la interfaz).
2. **Hilo y CPU**: Aunque los hilos son una característica de los procesos, la CPU juega un papel crucial en la ejecución de hilos. Los núcleos de la CPU ejecuta hilos, y los procesadores modernos están diseñados para manejar múltiples hilos simultáneamente. Algunas CPUs pueden manejar varios hilos por núcleo mediante tecnologías como Hyper-Threading, que permite que un solo núcleo ejecute más de un hilo a la vez.

En resumen, un hilo es una característica de los procesos que permite la ejecución concurrente de tareas dentro de un mismo proceso. La CPU proporciona el hardware necesario para ejecutar estos hilos y puede ejecutar múltiples hilos en paralelo, especialmente en sistemas con múltiples núcleos o tecnologías de simultaneidad.

Respuesta al ejercicio: Porque tengo muchos hilos dentro del proceso y cada hilo ejecuta en un núcleo distinto, por lo tanto el SO acumula todos los tiempos de CPU y los suma.

---

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

**user < real:** por la demora del trap y del inverso al trap (volver al usuario) añadida al tiempo de usuario
porque el sistema operativo antes de la ejecución del programa hace de resource manager.
Ya en las syscalls gastas algo

**user = real:** procesos de 1 solo hilo sin system calls (con ínfima cantidad de syscalls).

**real < user:** proceso multihilo, cada hilo ejecuta en un core distinto por lo tanto se suman todos para dar con el tiempo usuario

---

**Ejercicio 4**. Un programa define la variable `int x=100` dentro 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 cambian de valor?
(c) Contestar nuevamente las preguntas si el compilador genera código de máquina colocando esta
variable en un registro del microprocesador.

a) Como en `fork` se hace una copia del programa original incluyendo el estado, entonces el valor de la variable x en el hijo será 100.
b) Una vez creados sus estados son “independientes”, por lo tanto no depende de si el que la cambia es padre o hijo, que hagan lo que quieran.
c) Sigue todo joya porque tanto en el parent como en el child se respaldan los registros al hacer Trap y Return From Trap

---

Ejercicio 5. Indique cuántas 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.

n = 1

```
printf("a\n");              Se crea un hijo, ambos ejecutan el
siguiente printf
fork();                     (a = 3)
printf("a\n");
```

n = 2

```
printf("a\n");              Se crea un hijo, tenemos 2 procesos, Se
crea otro
fork();                     hijo, tenemos 4 procesos más por lo tanto
printf("a\n");              4 printf más
fork();                     (a = 7)
printf("a\n");
```

n = 3

```
printf("a\n");              Se crea un hijo, 2 procesos (a = 3), se
crea otro
fork();                     hijo, 4 procesos (a = 7), se crea otro hijo
printf("a\n");              8 procesos (a = 15)
fork();
printf("a\n");
fork();
printf("a\n");
```

En general dependiendo la cantidad de fork() que tengamos en el código se ejecutarán 2^n procesos (n cantidad de fork()). Luego, la cantidad de veces que se imprime “a” es 2^(n+1) - 1

Para usarlo en código c debemos de usar fflush() porque hacer una syscall es muy caro, entonces esta función lo que hace es mandar lo que haya para imprimir directamente, ya que se trata de minimizar el acceso al kernel, se ocasiona interferencia porque el printf no son directas las escrituras en stdout.

Desde que haces printf hasta que haces fork, no se imprime nada. Se acumula en un buffer interno, al multiplicar cada vez los procesos hay mas a's esperando por salir, cada proceso acumula las a … 4 a con 8 procesos = 32 a.

---

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

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

Ninguna, lo que hace es correr el programa ubicado en "/bin/date" con los argumentos "-R" y execv reemplaza la imagen del programa con la nueva, por lo que lo que está después de esta llamada nunca se ejecutará.

---

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);
        }
}
```

Primer programa:

- El programa toma dos argumentos:
  - Un entero argc.
  - Un arreglo de punteros que apuntan a cadenas.
- Primero aparece una guarda que se fija si el entero es mayor a 0 luego de ser decrementado en 1.
- En caso de dar negativo el programa no hace nada.

- En caso de ser positivo, la posición indicada por argc se establece en NULL y se llama a execvp con argv[0] como primer parámetro y argv como segundo parámetro.

Finalmente: El programa toma el tamaño de un arreglo y el arreglo en si, el cual contiene punteros a cadenas que indican el nombre del programa y luego sus argumentos, esto se deduce al ver la llamada a execvp quien como argumentos pide el nombre del programa para buscarlo en los directorios y además un arreglo que contenga el nombre, sus argumentos y un puntero a NULL en su última posición.

Segundo programa:

- El programa toma dos argumentos:
  - Un entero argc (tamaño del arreglo).
  - Un arreglo de punteros que apuntan a cadenas.
- Primero aparece una guarda que verifica que el arreglo contenga más de un elemento, caso contrario se retorna 0 y se termina la ejecución.
- En el caso de tener más de un elemento, se crea un hijo:
  - Si el hijo no se crea correctamente se retorna -1 indicando error.
  - Si el hijo se crea correctamente se retorna 0 y se acaba la ejecución (DEL HIJO) luego pasa al padre, quien asigna NULL a la anteúltima posición (pisando el último argumento).
    y llama a execvp con el programa nuevo.

---

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

```
LENO);
t", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);
amá salgo por un archivo!");
```

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

1) El printf siempre tira al stdout, al tener un open(file) el primer lugar libre que encuentra es el stdin en el open(file). Es rarito porque open puede fallar, open abre el primer fd libre, no podemos hacer control de errores.

2) El dup duplica fd en el file descriptor más cercano que encuentra, el más cercano libre, y el más cercano es el que abrimos en salida.txt. O sea que el dup() esta por si el open falla.

dup() nos ofrece un mejor manejo de archivos y redirección de entrada y salida, ya que tener que abrir y cerrar manualmente archivos en un contexto más complejo aumenta la probabilidad de cometer errores.

---

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

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

La bomba fork() lo que hace es crear procesos nuevos todo el tiempo, esto implica de cada uno de ellos necesiten memoria para guardar su contexto, registros, variables, etc. Eventualmente esto nos llena la memoria rompiendo todo. Puede ocurrir que se maten otros procesos para liberar memoria o el sistema se vuelve inoperable debido al consumo de recursos que tiene este proceso.

Se puede limitar la cantidad de procesos que se pueden correr con ulimit.
ulimit
ulimit -a
ulimit -u 32 “puedo hacer 32 forks()” podes bajarlo pero no subirlo.

---

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 cómo funciona (o no) el Sistema Operativo si
se quita solo una de las cuatro flechas.

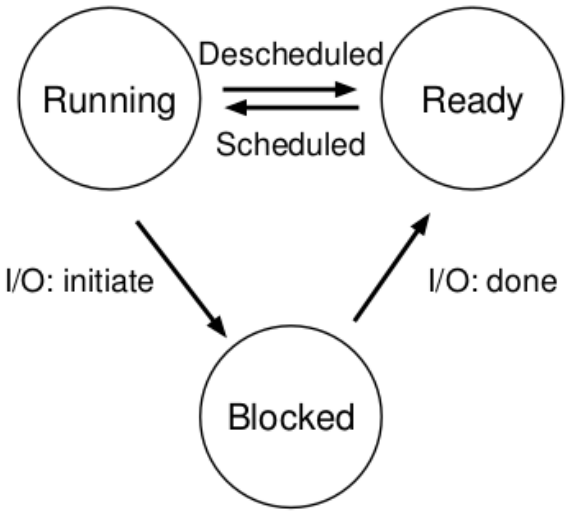


Figure 4.2: **Process: State Transitions**

Sacamos la flecha de DESCHEDULED:

- Si un proceso no tiene más operaciones de I/O, no va a terminar de ejecutar hasta el final, de esta forma podemos monopolizar el procesador. No habría time sharing.

Sacamos la flecha de SCHEDULED:

- No se ejecuta ningún proceso nunca, no hay nadie quien decida qué proceso corre o no.
  Todos los procesos nacen en READY

Sacamos la flecha de I/O: INITIATE:

- Si el proceso NECESITA de I/O este se quedará esperando mientras tiene el poder de la cpu, cuando podríamos compartirlo con otros procesos.

Sacamos la flecha de I/O: DONE:

- Un proceso que necesita de I/O nunca nos dará respuesta, se quedará bloqueado siempre.

---

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

```
struct trapframe {
```

```
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;     // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

---

Ejercicio 12.
Verdadero o falso. Explique.

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

- Un proceso que no tarda nada, no tiene syscall, no toca el kernel, pero a su vez se entorpece con otros procesos ajenos que se estaban ejecutando al mismo tiempo. Porque hay time sharing entre muchos procesos.

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

- El proceso 1 y 2 pueden estar usando la misma memoria virtual porque cada proceso piensa que tiene toda la RAM física para ellos.

(c) Para guardar el estado del proceso es necesario salvar el valor de todos los registros del micropro-
cesador. (V)

- Se necesita guardar todo el contexto del proceso, si dejo algo puede ser pisado o entorpecer otros procesos.

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

- No todas, las privilegiadas no pude.

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

- Quizás es el único proceso ejecutando.

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

- Es por decisión de diseño.

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

- Para eso se separaron.

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

- Se puede ejecutar primero el hijo y luego el padre, o al revez. De hecho wait() nos ayuda en el caso en el cual el padre se ejecuta y no el hijo.

(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. (F)

- Si se puede, redirigir la salida del hijo a un fd que el padre haya creado.
- Con return y el waitpid()
- Son independientes pero el inicio y el fin están conectados.

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

- Si falla se ejecuta lo que está después.

(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()`. (V)

- Una vez que el hijo termina, queda en estado zombie hasta que el padre pueda leer su estado de salida y así terminar la ejecución.

# 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.

FIFO o FCFS (First Come First Serve) o la cola del super= First in First out

| 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 | 80 | 85 | 90 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C | C | C | B | B | B | B | A | A | A | A | A | A | — | — | — | —- | — | — | — |

SJF (Shortest Job First)

| 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 | 80 | 85 | 90 | 95 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C | C | C | B | B | B | B | A | A | A | A | A | A | | | | | | | |

---

**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 $T_{re\ sponse}$ 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 | | | | |

**STCF** (shortest time-to-completion first)(como SJF pero los proceso pueden llegar en cualquier momento)

| **Tiempo** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Running** | $B^3$ | $B^2$ | $B^{1*}$ | $A^4$ | $C^{1*}$ | $A^3$ | $A^2$ | $A^1$ | | | | | | | | | | | | |
| **Arribos** | B | | A | | C | | | | | | | | | | | | | | | |
| **Ready** | | | A | | A | | | | | | | | | | | | | | | |

* Se compara con A y gana B por tener menos $T_{CPU}$, entonces sigue ejecutando B.
** Igual que antes, C tiene menos $T_{CPU}$ que A, C gana por política STCF. A queda en Ready esperando a que termine C.

Entro de una

Desempate: El que estaba corriendo que siga corriendo. Es caro cambiar entre procesos.

| **Proceso** | **$T_{arrival}$** | **$T_{CPU}$** | **$T_{firstrun}$** | **$T_{completion}$** | **$T_{turnaround}$** | **$T_{response}$** |
|---|---|---|---|---|---|---|
| A | 2 | 4 | 2 | 7 | 5 | 0 |
| B | 0 | 3 | | | | |
| C | 4 | 1 | | | | |

**RR** (Round Robin)(Q=2)(los procesos solo se ejecutan durante 2 quantos, luego van al final de la cola)(FIFO con quanto)

| **Tiempo** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Running** | $B^3$ | $B^2$ | $A^4$ | $A^3$ | $C^{1*}$ | B | $A^2$ | $A^1$ | | | | | | | | | | | | |
| **Arribos** | B | | A | | C | | | | | | | | | | | | | | | |
| **Ready ¿Cola?** | | | B | B | B<br>A | A | | | | | | | | | | | | | | |
| **Qs Restantes** | B:3 | B:2 | A:4 | A:3 | C:<br>1 | B:1 | A:2 | A:1 | | | | | | | | | | | | |

*C tiene **$T_{CPU}$** = 1, sólo dura 1 quanto. Había simultaneidad entre B y C, nos decidimos por el C por política de prioridad: el proceso que entra se ejecuta. Esta política debe ser consistente para cada caso recurrente.

Posibles políticas para tratar el desempate:

- Podrías darle prioridad a procesos que recién aparecen o arriban

- Decido ejecutar el que ejecute antes. Rezando que está cacheado o algo asi
- Decido ejecutar el C porque no se ejecutó ninguna vez, para ser justo.

---

**Ejercicio 15**

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

| | **Batch/interactive** | **¿Necesita saber $T_{cpu}$?** |
|---|---|---|
| **FIFO** | batch | No |
| **SJF** | batch | Si |
| **RR** | Interactivo | No |
| **STCF** | 50/50 | Si |
| **MLFQ** | Interactivo | No |

Interactivo significa tiempo de respuesta corto

*¿Por qué es 50/50 el STFC?* ⟶ No corta por quanto (no es interactivo en ese sentido), pero sí es interactivo si llegan procesos cortos.

---

**Ejercicio 16**

Considere los siguientes procesos que mezclan ráfagas de CPU con ráfagas de IO.
Realice el diagrama de planificación para un planificador RR (Q=2). Marque bien cuando el proceso
está bloqueado esperando por 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 |

**$T_{CPU}$** total de cada proceso: A=6, B=14, C=6.

**RR** (Round Robin)(Q=2)(los procesos solo se ejecutan durante 2 quantos, luego van al final de la cola)(FIFO con quanto)

| | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ¹ | $B^8$ | $B^7$ | $A^1$ | $B^6$ | $B^5$ | $C^2$ | $C^1$ | $B^4$ | $B^3$ | $A^2$ | $A^1$ | $B^2$ | $B^1$ | $C^1$ | $B^6$ | $B^5$ | **$A^1$** | $B^4$ |
| | C | C | C | A | A | A | A | C,A | C | C | C | C,A | A | B,A | A,C | C | C | C |
| | | | | C | | | | | A | | | | C | | B | A | | |
| | A | A | B | C | C | B | B | | A | B | B | | C | | | | B | |
| | | | | | | | | | | | | | | | | | | A |

Política: Si un proceso A termina su quanto al MISMO tiempo que arriba un B, el B va primero en la cola Ready.

---

**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:

c

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

convención: $Q_n$ (<--- )

| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **C** | | **D** | | | **E** | | | | | | | | | | |
| BA | CB**A** | CBA | DCB **A** | DC**B** | DCB | EDC **B** | ED**C** | ED**C** | E**D** | E**D** | **E** | **E** | | | |
| | | | | A | A | A | A | A | CA | CA | CA | CA | EC**A** | EC**A** | EC**A** |
| | | | | | | | | | | | | | | | |

En negrita están marcados los procesos que se ejecutan en cada tiempo.
Se subrayó y coloreó cada proceso en el tiempo que termina.
Si un proceso ejecuta la cantidad (n) de quantos correspondiente a cada cola de prioridad, baja a la siguiente cola de prioridad. Esta ejecución puede ser incontinua. ->política MLFQ, acumula las ejecuciones del proceso en la cola para mandarlo a una cola de menor prioridad.

---

**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. Falso, se puede tomar el control de otras formas como por ejemplo I/O interrupciones.

b) Entre las políticas por lote FCFS y SJF, hay una que siempre es mejor que la otra respecto a

Tturnaround . Verdadero

| | | |
|---|---|---|
| A | B | C |

En este ejemplo:
$T_{arrival}$ A = 0 $T_{TIME}$ A = 100
$T_{arrival}$ B = 10 $T_{TIME}$ B = 10
$T_{arrival}$ C = 10 $T_{TIMEl}$ C = 10

En SJF el promedio del turnaround es de (20 - 10) + (30 - 10) + (120 - 0) / 3 = 50
Pero en FCFS tenemos que esperar a que termine A para ejecutar B y luego C, es decir, (100 - 0) + (110 - 10) + (120 - 10) / 3 = 103,33

c) La política RR con quanto = ∞ es FCFS. Falso, round robin no tiene criterio para seleccionar los procesos. Solo le asigna un tiempo a cada uno.

d) MLFQ sin priority boost hace que algunos procesos pueden sufrir de starvation (inanición). Verdadero, cuando hay procesos I/O, cuando entran muchos procesos, etc.

e) En MLFQ acumular el tiempo de CPU independientemente del movimiento entre colas evita hacer

trampas como yield() un poquitito antes del quantum. Verdadero, esto permite que un proceso no suelte el cpu siempre antes del cuanto para tomar el control, si hace esto la próxima ejecución se corta enseguida y baja de prioridad.