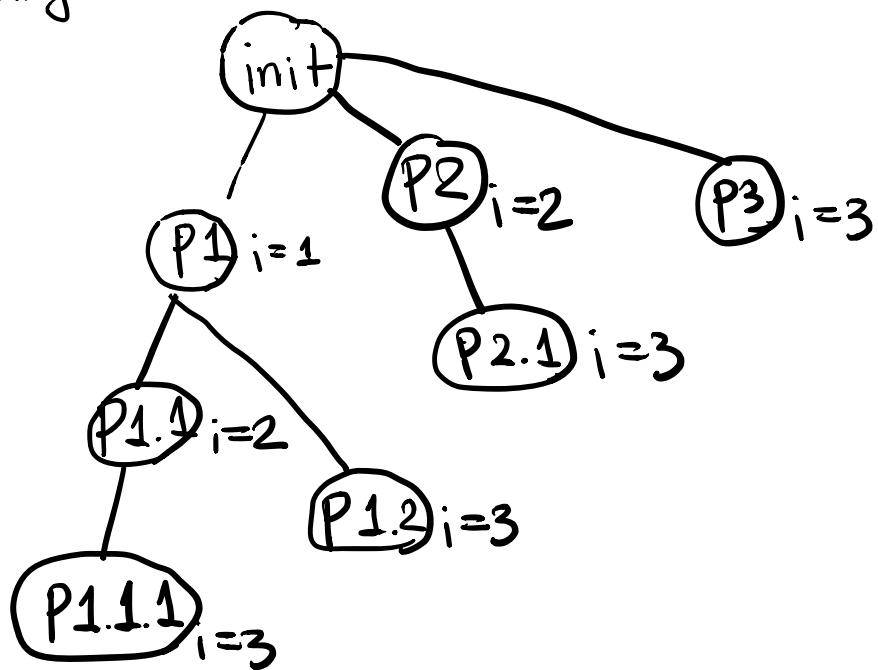


1.- Dado el siguiente programa ejecutado bajo UNIX:

```
void main(int argc, char *argv[]){
    int i;
    for(i=1; i<=argc; i++)
        fork();
    ...
}
```

- a) Dibuje el esquema jerárquico de procesos que se genera para $\text{argc}=3$
b) ¿Cuántos procesos se crean para $\text{argc}=n$?

a) $\text{argc} = 3$.



b) $n=3$, se crean 7 procesos

$\text{argc}=n$, se crearán $2^n - 1$ procesos

2.- Considere el siguiente código:

```
int varGlobal;

void main() {
    int varLocal=3;
    pid_t pid;

    varGlobal=10;
    printf("Soy el proceso original. Mi PID es %d\n", getpid());
    fflush(NULL);

    pid = fork();
    if (pid ==-1) {
        perror("Error en fork()\n");
        exit(-1);
    }
    if (pid == 0 ) {
        // CODIGO DEL PROCESO HIJO
        varGlobal = varGlobal + 5;
        varLocal = varLocal + 5;
    }
    else {
        // CODIGO DEL PADRE: pid contiene el pid del hijo
        wait(NULL);
        varGlobal = varGlobal + 10;
        varLocal = varLocal + 10;
    }
    printf("Soy el proceso con PID %.d. Mi padre es %.d Global: %.d Local %.d\n",
           getpid(), getppid(),varGlobal, varLocal );
}
```

Asumiendo que el proceso original tiene PID=100 y es hijo del proceso `init` (PID=1), indica qué se mostrará por pantalla al ejecutar el código. ¿Es posible que los valores finales de las variables varíen de una ejecución a otra en función del orden de planificación? ¿Puede cambiar el orden en el que se muestran los diferentes mensajes por pantalla?

Por pantalla,

Soy el proceso original. Mi PID es 100

Soy el proceso con PID 101. Mi padre es 100. Global:15 Local:8

Soy el proceso con PID 100. Mi padre es 1. Global:20 Local:13

No se comparten las variables globales ni locales.

No, porque el padre espera a que termine el hijo siempre. El orden se mantiene.

3.- Considere el siguiente código:

```
int a = 3;
void main() {
    int b=2;
    for (i=0;i<4;i++) {
        p=fork();
        if (p==0) {
            b++;
            execvp("comando" ...);
            a++;
        }
        else {
            wait();
            a++;
            b--;
        }
    }
    imprime(a,b);
}
```

a) ¿Cuántos procesos se crean en total? (sin contar el padre original) ¿Cuántos coexisten en el sistema como máximo?

b) ¿Qué se imprimirá al mostrar los valores de a y b?

a) Se crean 4 procesos.

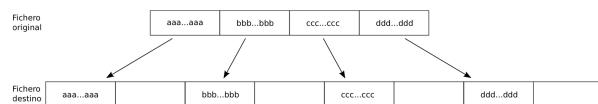
Al hacer execp(deja de tener el código), no continúa ejecutando el hijo. Solo él pradera ejecuta el bucle for.

El número de procesos coexisten simultáneamente como máximo es 2.

b) a=7, b=-2. Se imprimirá solo los del padre.

4.- Un programador poco avezado pretende hacer una aplicación que realice copias intercaladas de ficheros en paralelo. El concepto de copia intercalada se ilustra en la figura: se copia el primer bloque entero, y se deja un bloque vacío. Se copia el segundo bloque del fichero origen al tercer y después para cada uno de los hijos en el orden de creación.

Responde a las siguientes preguntas, suponiendo que la **prioridad es para el proceso padre**, y así sucesivamente.



- a) Indica el contenido del array **buf** inmediatamente después de la ejecución de las líneas 29 y 31. Justifica tu respuesta.
 b) Sea sistema de ficheros de tipo Linux (nodos-i), con 3 punteros directos y un indirecto simple, tamaño de bloque de 4KiB (4096 bytes) y 4bytes por puntero. Dibuja un posible

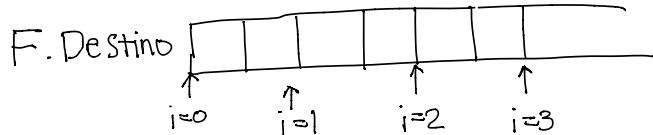
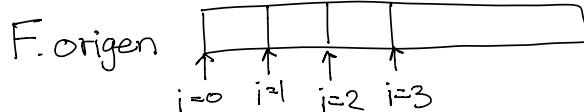
Versión del 12 de marzo de 2018

- 2 -

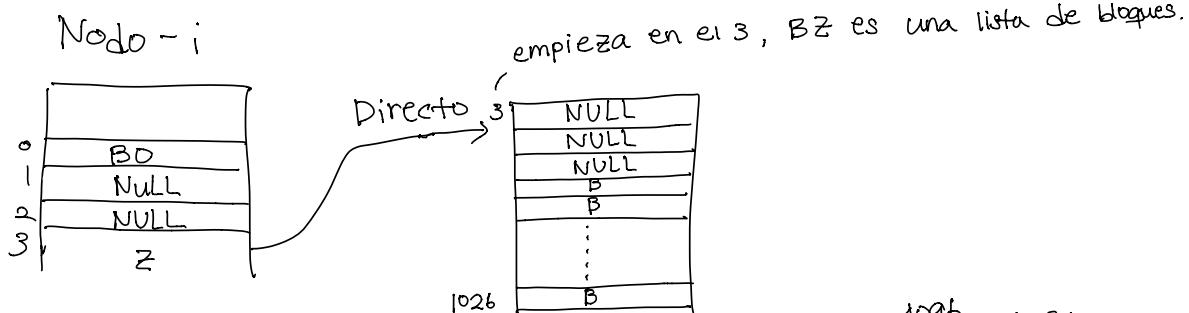
El código de su programa para la copia intercalada de un fichero de 4 bloques mediante 4 procesos, es el siguiente:

```

1 #define BLOCK 4096
2 char buf[BLOCK] = "xxxxxxxx...xxxxx";
3
4 void copia_bloque(int fdo, int fdd) {
5   read(fdo,buf,BLOCK);
6   write(fdd,buf,BLOCK);
7 }
8
9 void main() {
10   pid_t pid;
11   int fdo,fdd;
12
13   fdo = open("Origen",O_RDONLY);
14   fdd = open("Destino",O_RDWR|O_CREAT|
15             O_TRUNC,0666);
16   for (int i=0; i < 4; i++) {
17     lseek(fdo,i*BLOCK,SEEK_SET);
18     lseek(fdd,2*i*BLOCK,SEEK_SET);
19     pid = fork();
20     if (pid==0){ //ijo
21       copia_bloque(fdo,fdd);
22       exit(0);
23     }
24   }
25 }
26
27 while (wait(NULL)!=-1) {
28
29   read(fdd,buf,BLOCK);
30   lseek(fdd,0,SEEK_SET);
31   read(fdd,buf,BLOCK);
32 }
```



El buf está escrito en el primer bloque.



Hay 1024 bloques porque $\frac{4096}{4} = 1024$
 Y en los bloques del resto tendrá NULL.



4. Se comparte el marcador del fichero entre los procesos

```
1 #define BLOCK 4096
2 char buf[BLOCK] = "xxxxxxxx...xxxxxx";
3
4 void copia_bloque(int fdo, int fdd) {
5     read(fdo,buf,BLOCK);
6     write(fdd,buf,BLOCK);
7 }
8
9 void main() {
10    pid_t pid;
11    int fdo,fdd;
12
13    fdo = open("Origen",O_RDONLY);
14    fdd = open("Destino",O_RDWR|O_CREAT|
15               O_TRUNC,0666);
16
17    for (int i=0; i < 4; i++) {
18        lseek(fdo,i*BLOCK, SEEK_SET);
19        lseek(fdd,2*i*BLOCK, SEEK_SET);
20
21        pid = fork();
22        if (pid==0){ hijo
23            copia_bloque(fdo,fdd);
24            exit(0);
25        }
26
27        while (wait(NULL)!=-1) {
28
29            read(fdd,buf,BLOCK);
30            lseek(fdd,0,SEEK_SET);
31            read(fdd,buf,BLOCK);
32        }
33
34    }
35}
```

Genera 4 procesos hijo pero no realiza la copia hasta que el padre acabe su ejecución. Cuando se acabe el padre, el descriptor está al final y no va a copiar nada. Cuando le toque al hijo, como el marcador compartido está en el final, va a copiar nada tampoco.

Para solucionar este problema, ponemos que cada proceso hijo tenga su propio marcador.
El cambio:

```
void copia_bloque (int i){  

    fdo = open ("Origen", O_RDONLY);  

    fdd = open ("Destino", ... );  

    lseek (fdo,i*BLOCK, SEEK_SET);  

    lseek (fdd, 2*i*BLOCK, SEEK_SET);  

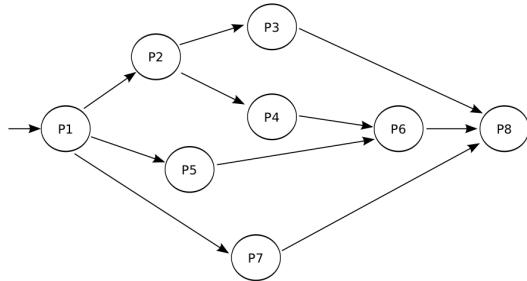
    read (...);  

    write(...);  

    close(fdo);  

    close(fdd);
```

5.- Use las llamadas `fork()`, `exec()`, `exit()` y `wait()` de UNIX para describir la sincronización de los ocho procesos cuyo grafo general de precedencia es el siguiente. Para ello escriba un función `main()` en la que se vayan creando los 8 procesos mediante llamadas a `fork()`, se ejecuten los binarios asociados a cada proceso (por ejemplo, `p1.out` para el proceso `p1`, etc.) y se respete la precedencia mostrada en la figura (por ejemplo, `p6` no puede comenzar hasta que no hayan acabado `p4` y `p5`).



```

int main(){
    pid1=fork();
    if(pid1==0){
        execlp(p1.out...);
        exit(-1);
    }

    wait.pid(pid1,null,0);
    pid2=fork();
    if(pid2==0){
        execlp("p2.out",...);
        exit(-1);
    }

    pid5=fork();
    if(pid5==0){
        execlp("p5.out",...);
        exit(-1);
    }

    pid7=fork();
    if(pid7==0){
        execlp("p7.out",...);
        exit(-1);
    }

    wait.pid(pid2,null,0);
    pid3=fork();
    if(pid3==0){
        execlp("p3.out",...);
        exit(-1);
    }

    pid4=fork();
    if(pid4==0){
        execlp("p4.out",...);
        exit(-1);
    }
}

wait.pid(pid4,null,0);
wait.pid(pid5,null,0);
pid6=fork();
if(pid6==0){
    execlp("p6.out",...);
    exit(-1);
}

wait.pid(pid3,null,0);
wait.pid(pid6,null,0);
wait.pid(pid7,null,0);

pid8=fork();
if(pid8==0){
    execlp("p8.out",...);
    exit(-1);
}

wait.pid(pid8,null,0);
// main
  
```

6.- Considere el siguiente código:

```
int fd = -1;
char buf1[4] = "aaaa";
char buf2[4] = "bbbb";

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, h1, NULL);
    pthread_create(&tid2, NULL, h2, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    close(fd);
}

void* h1() {
    fd = open("prueba", O_RDWR | O_CREAT | O_TRUNC
              , 0666);
    write(fd, buf1, 4);
}

void* h2() {
    while (fd == -1) {};
    write(fd, buf2, 4);
}
```

porque el marcador
del fichero es
una var. global y
lo comparten H1 y H2

Indique si cada una de las siguientes afirmaciones es cierta o falsa justificando la respuesta:

- a) El hilo 2 (función h2) nunca saldrá del bucle while inicial.
- b) La escritura del hilo 2 (función h2) producirá un error por no haber abierto antes el fichero.
- c) El contenido final del fichero prueba será o bien, aaaa o bien bbbb; ninguna otra alternativa es posible.
- d) La llamada a close() del programa principal no debería devolver -1 (esto es, no debería producir ningún error).

- a) Falsa. El hilo1 cambia el valor de fd. ✓
- b) Falsa. Entre hilos, se comparten el fichero ✓
- c) Falsa. Se comparte el fd. El contenido será aaaabbbb o bbbbaaaa. o más cosas ✓
- d) cierto. ✓ Excepto cuando el open de error, no se puede cerrar el fichero

7.- En un sistema monoprocesador los siguientes trabajos llegan a procesarse en los instantes indicados. ¿Cuáles son los tiempos de retorno (turnaround) y de espera para cada uno de ellos, los tiempos de retorno y de espera promedios, así como la productividad (throughput) del sistema, aplicando las diferentes estrategias de planificación listadas? Aplique los algoritmos de planificación sin expropiación y base las decisiones en la información de que se dispone en el momento de tomarlas.

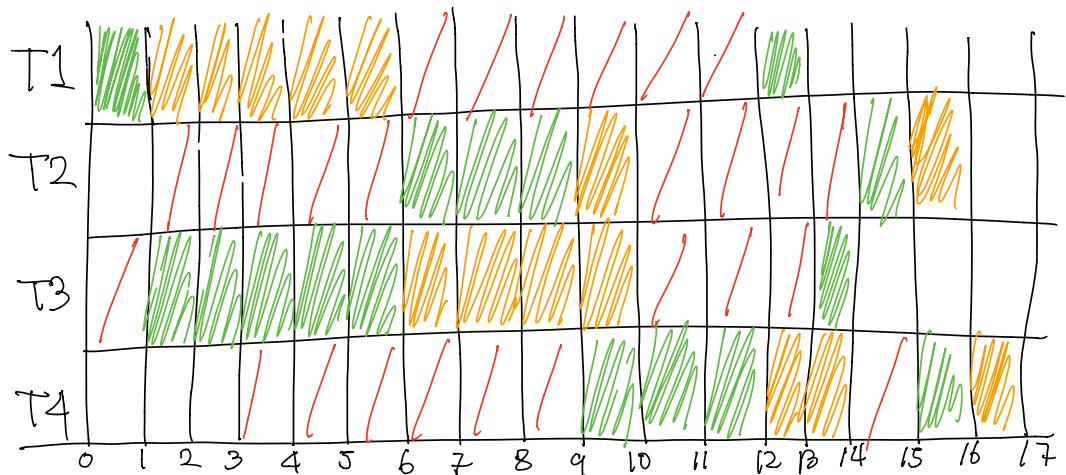
Trabajo	Llegada	CPU	E/S	CPU	E/S
1	0	1	5	1	
2	1	3	1	1	1
3	0	5	4	1	
4	3	3	2	1	1

- a) FCFS: Primero en llegar, primero en pasar (sin expropiación)
- b) SPN: Primero el de menor tiempo de UCP siguiente
- c) RR: Prioridad circular con cuanto = 3
- d) RR: Prioridad circular con cuanto = 1

T. Ejecución

T. Bloqueo

T. Espera (Bloques por espera de E/S)



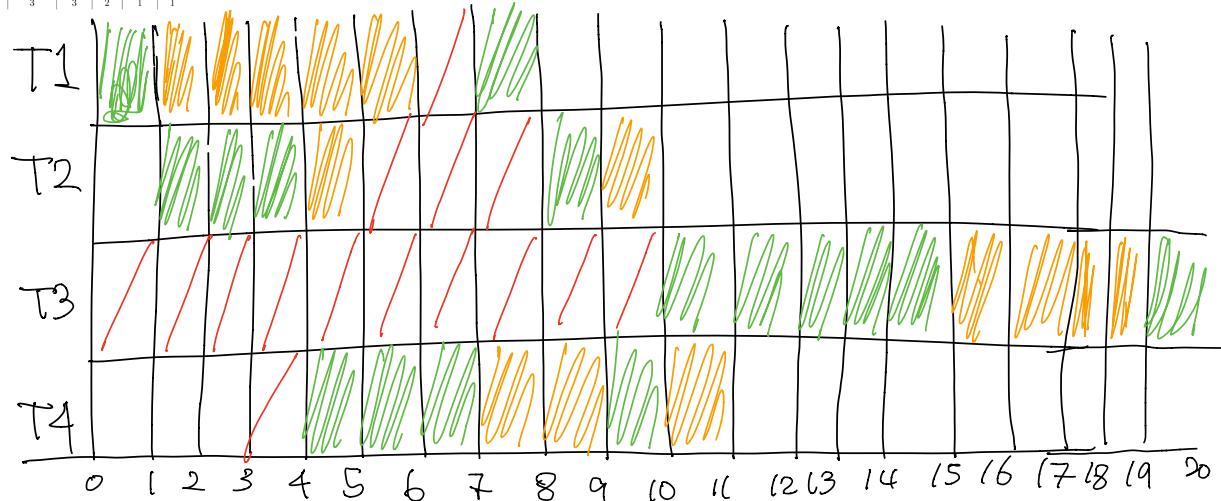
FCFS	T1	T2	T3	T4	Media
------	----	----	----	----	-------

T.espera	6	9	4	7	6.5
----------	---	---	---	---	-----

T.ej.	13	15	14	14	14
-------	----	----	----	----	----

Trabajo	Llegada	CPU	E/S	CPU	E/S
1	0	1	1	1	1
2	1	3	1	1	1
3	0	5	4	1	1
4	3	3	2	1	1

$$\text{throughput} = \frac{n^{\circ} \text{ tareas}}{t. \text{ tardado total}} = \frac{4}{17} = 0.23$$



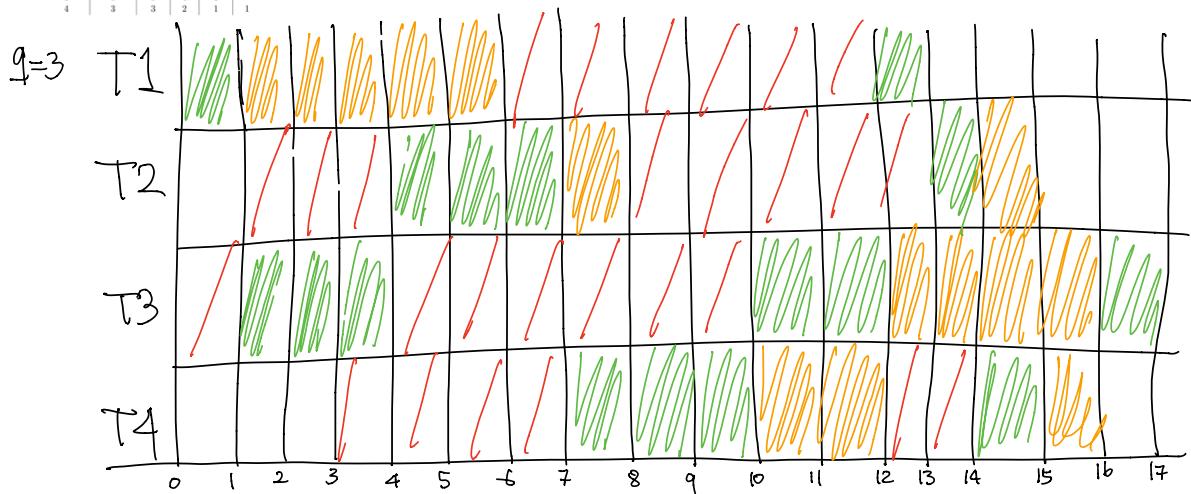
SPN	T1	T2	T3	T4	Media
-----	----	----	----	----	-------

T.espera	1	3	10	1	3.75
----------	---	---	----	---	------

T.ej	8	9	20	8	11.25
------	---	---	----	---	-------

$$\text{Trabajo} \quad \begin{array}{|c|c|c|c|c|c|c|} \hline & \text{Llegada} & \text{CPU} & \text{E/S} & \text{CPU} & \text{E/S} \\ \hline 1 & 0 & 5 & 1 & 1 & 1 \\ 2 & 1 & 3 & 1 & 1 & 1 \\ 3 & 0 & 5 & 4 & 1 & 1 \\ 4 & 3 & 3 & 2 & 1 & 1 \\ \hline \end{array}$$

$$\text{throughput} = \frac{4}{20} = 0'2$$

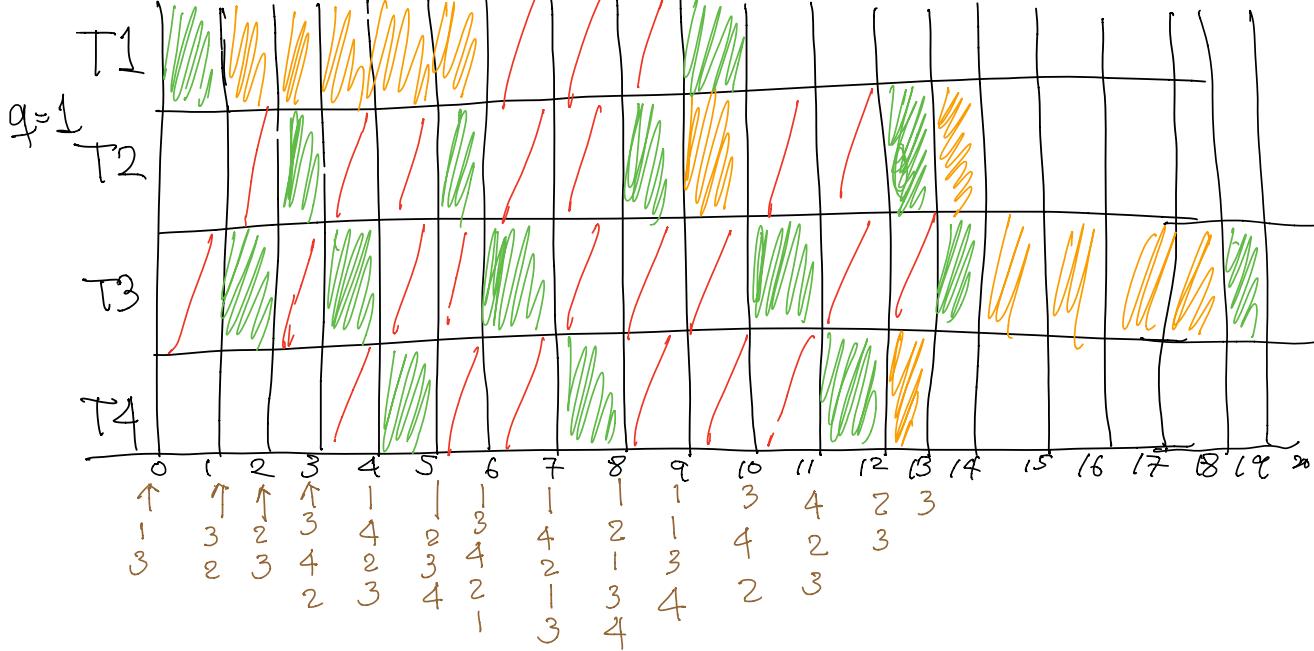


$RR_{q=3}$	T1	T2	T3	T4	Media
T. espera	6	8	7	6	6.75

T. ej 13 14 17 13 14.25

Trabajo	Llegada	CPU	E/S	CPU	E/S
1	0	1	5	1	1
2	1	3	1	1	1
3	0	5	4	1	1
4	3	3	2	1	1

$$\text{throughput} = \frac{4}{17} = 0'23$$



	T1	T2	T3	T4	Media
T. espesa	3	7	9	6	6'25
T. ej	10	13	19	10	13

$$\text{Throughput} = \frac{4}{19} = 0'21$$