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) Dadas las muestras podemos asumir que el cpu tiene 2 núcleos, chequeando los tiempos de usuario.
b) Puede ser por varios factores. Podría ser porque no está tomando en cuenta el systime o porque algún otro proceso se ejecutó entre medio.
c) Los enumeremos de 1…7 de arriba hacia abajo:
   1) Se ejecuta el programa, pasa 2.44s en usermode y 2.56s de walltime
   2) Lo mismo, pero como son dos cálculos el programa usa los dos núcleos
   3) Idem 1, pero el walltime es más alto, probablemente algo se ejecutó entre medio
   4) Idem 2, se asignan dos cálculos a cada procesador.
   5) Dos núcleos para tres cálculos, pero walltime es menor a lo que debería??
   6) Probablemente se usó un solo núcleo para los dos cálculos, por eso el walltime es tan alto.
   7) 4 cálculos, pero un núcleo trabaja en 2 y los otros 2 se turnan con otro núcleo.

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
```

Una posible respuesta es que el programa divida la ejecución del programa en varios hilos del procesador, que cuando se suma el tiempo da mayor al walltime.

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

casos:
user < real : en programas que queden esperando señales I/O del usuario, donde por ejemplo se hace un cálculo y queda esperando por confirmación del usuario si quiere mostrarlo por pantalla.

user = real : en programas que sean exclusivamente de ejecución, por ejemplo calcular pi.

user > real : en programas que usen multithreading

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) asumiendo que hace fork luego de asignarle el valor, entonces x = 100 tanto en el padre como en el hijo.
b) Nada, son dos procesos independientes, no tienen injerencia uno en el otro.
c) ????

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

Este programa imprime 1 + 2 + 4 + 8 = 15 (a)
Es decir dados n forks hace ∑(i=0) hasta n de 2^n

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

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

Imprime 0 "a" asumiendo que execv se ejecute correctamente. Si execv falla entonces se imprime una vez.

Ejercicio 7. Indique que hacen estos programas.

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

        return 0;
}
```

```c
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.

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

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

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

1) Cierra la salida estándar()

abre el archivo salida.txt en el filedescriptor 1, porque es el que más arriba está libre.\
printf imprime en el filedescriptor 1(siempre según la man page)

2) Abre el archivo salida.txt

cierra la salida estándar
dup(duplica el file descriptor fd en el primer filedescriptor libre)
printf imprime en salida.txt
Sí hacen lo mismo.

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

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

Los sistemas operativos tienen OOM killer (out of memory killer) que se encarga de que un proceso no acapare toda la memoria. En este caso no es un solo proceso si no que son

muchos, esto puede ser mitigado con un ulimit que determina cuántos procesos puede crear el usuario.
$ ulimit -a
.
.
.
max user processes (-u) 127612
.
.
.

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.

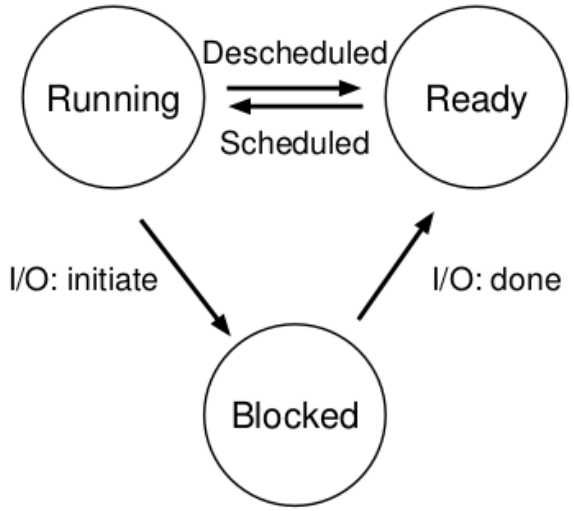

Figure 4.2: **Process: State Transitions**

1. Si se elimina la flecha de Running a Ready (Descheduled):

- **Escenario**: Un proceso que está en ejecución (Running) no podría pasar al estado de listo (Ready) si es desprogramado por el planificador.
- **Impacto en el Sistema Operativo**: Los procesos que están ejecutándose no podrían ser preemptados (interrumpidos) y volver al estado listo. Esto podría llevar a un monopolio de CPU por un solo proceso, causando que otros procesos no puedan ser ejecutados, resultando en un sistema ineficiente o en "starvation" para otros procesos.

2. Si se elimina la flecha de Ready a Running (Scheduled):

- **Escenario**: Un proceso que está listo para ejecutarse no podría ser seleccionado por el planificador para pasar al estado de ejecución.

- **Impacto en el Sistema Operativo**: Ningún proceso en la cola de procesos listos podría ser ejecutado. Esto detendría toda la ejecución en el sistema, ya que los procesos no podrían avanzar del estado de listo a ejecución, causando una parada total en el sistema operativo.

3. Si se elimina la flecha de Running a Blocked (I/O initiate):

- **Escenario**: Un proceso en ejecución no podría pasar al estado bloqueado cuando inicia una operación de entrada/salida (I/O).
- **Impacto en el Sistema Operativo**: Los procesos que requieren realizar operaciones de I/O no podrían bloquearse y esperar hasta que la operación de I/O se complete. Esto podría llevar a un estado de ineficiencia, donde el proceso sigue ocupando la CPU mientras espera la operación de I/O, afectando el rendimiento general del sistema.

4. Si se elimina la flecha de Blocked a Ready (I/O done):

- **Escenario**: Un proceso que está bloqueado no podría pasar al estado listo después de que se complete su operación de I/O.
- **Impacto en el Sistema Operativo**: Los procesos que terminan su operación de I/O quedarían atrapados en el estado bloqueado y no podrían ser programados para ejecución. Esto resultaría en una acumulación de procesos bloqueados que nunca regresarían a la ejecución, causando que el sistema no progrese en el manejo de procesos que dependen de I/O.

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.

Parte apilada por el hardware: `cs`, `eip`, `eflags`, `esp` (en caso de cambio de privilegio), `ss` (en caso de cambio de privilegio).
Parte apilada por el software: `edi`, `esi`, `ebp`, `oesp`, `ebx`, `edx`, `ecx`, `eax`, `trapno`, `err`, `ds`, `es`, `fs`, `gs`.

Ejercicio 12. Verdadero o falso. Explique.
(a) Es posible que user+sys < real. **V**
(b) Dos procesos no pueden usar la misma dirección de memoria virtual. **Fo V?**
(c) Para guardar el estado del proceso es necesario salvar el valor de todos los registros del microprocesador. **V**
(d) Un proceso puede ejecutar cualquier instrucción de la ISA. **F, son instrucciones privilegiadas**
(e) Puede haber traps por timer sin que esto implique cambiar de contexto.**V**
(f) fork() devuelve 0 para el hijo, porque ningún proceso tiene PID 0.**F, el proceso 0 es usualmente idle**
(g) Las syscall fork() y execv() están separadas para poder redireccionar los descriptores de archivo.**V??**

(h) Si un proceso padre llama a exit() el proceso hijo termina su ejecución de manera inmediata. **F queda huerfano**
(i) Es posible pasar información de padre a hijo a través de argv, pero el hijo no puede comunicar información al padre ya que son espacios de memoria independientes. **F**
(j) Nunca se ejecuta el código que está después de execv().**F**
(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 Tarrival en 0 para todos y Tcpu de 30, 20 y 10 respectivamente. Dibujar la línea de tiempo para las políticas de planificación FCFS y SJF. Calcular el promedio de Tturnaround y Tresponse para cada política.

FCFS	0	10	20	30	40	50	60
Tarrival	ABC						
Tcpu	A	A	A	aB	B	bC	Cfin

Tturnaraound (A)=30-0=30
Tturnaraound (B)=50-0=50
Tturnaraound (C)=60-0=60
Tturnaraound (promedioABC)=(30+50+60)/3=46,66

Tresponse (A)=0-0=0
Tresponse (B)=30-0=30
Tresponse (C)=50-0=50
Tresponse (promedioABC)=(0+30+50)/3=26,67

SJF	0	10	20	30	40	50	60
Tarrival	ABC						
Tcpu	C	cB	B	bA	A	A	Afin

Tturnaraound (A)=60-0=60
Tturnaraound (B)=30-0=10
Tturnaraound (C)=10-0=0
Tturnaraound (promedioABC)=(60+30+10)/3=33,33

Tresponse (A)=30-0=30
Tresponse (B)=10-0=10
Tresponse (C)=0-0=0
Tresponse (promedioABC)=(30+10+0)/3=13,33

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				

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.

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 A	A														
Qs Restantes	B: 3	B: 2	A: 4	A: 3	C: 1	B: 1	A: 2	A: 1												

*C tiene $\mathbf{T_{CPU}}$ = 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)

Tiempo	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
Running (CPU)	A^3	A^2	C^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	B^3	C^2	**C^1**	B^2	B^1
Blocked (I/O)				C	C	C	A	A	A	A	C, A	C	C	C	C, A	A	B, A	A, C	C	C	C					
Arribos	A	C	B				C					A				C		B	A			C				
Ready Cola		C	**B A**	A	A	B	C	C	B	B		A	B	B		C				B		C	B	B		
Termino																					A				C	

Política: Si un proceso A termina su cuanto 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 (<---)

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Q_1	**A**	**B**	**C**		**D**			**E**													
Q_2		A	BA	CB **A**	CB A	DC B**A**	DC **B**	DC B	ED C**B**	ED **C**	ED **C**	E**D**	E**D**	**E**	**E**						
Q_4							A	A	A	A	A	CA	CA	CA	CA	EC **A**	EC **A**	EC **A**	EC **A**	E**C**	**E**
Q_8																					

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. ->politica 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. **F**

b) Entre las políticas por lote FCFS y SJF, hay una que siempre es mejor que la otra respecto a Tturnaround. **V, en general**

c) La política RR con quanto = ∞ es FCFS. **V**

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

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