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.

Estacionamiento(GRATIS):

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

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.

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
```

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 cant 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 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) 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 cuantas letras “a” imprime este programa, describiendo su funcionamiento.

```
printf("a\n"); //1 a
fork();        // 2 procesos
printf("a\n"); // +2a, 3 a totales
fork();        //2*2 procesos
```

```
printf("a\n"); //+4a, 7 a totales
fork();        // 2*2*2 procesos
printf("a\n"); //+8a, 15 a totales
```

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

Mejor explicado:
Paso a paso:

1. **Inicio:**
   - Hay 1 proceso (P0).
   - Este proceso ejecuta `printf("a\n")`, lo que imprime "a".
2. **Después del primer :**
   - `fork()` crea un nuevo proceso. Ahora hay 2 procesos en total: P0 y P1.
   - Ambos procesos (P0 y P1) ejecutan `printf("a\n")`, lo que imprime "a" dos veces (una vez en cada proceso).
3. **Hasta ahora:**
   - "a" se ha impreso 1 (inicial) + 2 (después del primer `fork()`) = 3 veces.
4. **Después del segundo :**
   - Cada uno de los 2 procesos (P0 y P1) crea un nuevo proceso. Ahora hay 4 procesos en total: P0, P1, P2, y P3.
   - Cada uno de los 4 procesos ejecuta `printf("a\n"` , lo que imprime "a" cuatro veces (una vez en cada proceso).
5. **Hasta ahora:**
   - "a" se ha impreso 3 (previo) + 4 (después del segundo `fork()`) = 7 veces.
6. **Después del tercer :**
   - Cada uno de los 4 procesos (P0, P1, P2, P3) crea un nuevo proceso. Ahora hay 8 procesos en total: P0, P1, P2, P3, P4, P5, P6, y P7.
   - Cada uno de los 8 procesos ejecuta `printf("a\n")`, lo que imprime "a" ocho veces (una vez en cada proceso).
7. **Hasta ahora:**
   - "a" se ha impreso 7 (previo) + 8 (después del tercer `fork()`) = 15 veces.

Para calcular el número total de impresiones de "a", debemos tener en cuenta que cada proceso (al final) imprimirá "a" en cada uno de los 4 puntos del código:

- **Total de procesos al final:** 8 (cada proceso es creado por cada `fork()`).
- **Número de impresiones por proceso:** 4 (una por cada `printf("a\n")` en el código).

Número total de impresiones:

8 procesos × 4 impresiones = 32 impresiones.

Entonces, el número total de impresiones de "a" es **32**.

!cualquier parecido con chat gpt es pura coincidencia

f(n)=(2^n) - 1

n =

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

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

Ninguna, ya que el execv() de funcionar correctamente, (toda syscall tiene posibilidad de fallar) no returnea a esta función, evitando que la línea del printf se ejecute

No debería imprimir ninguna 'a' (si es que no falla el comando), ya que el comando 'execv(args[0], args)' reemplaza el programa del proceso actual por el del programa que ejecuta como argumento en args[0]. Si falla, imprime 1 'a'.

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.

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

1)	2)

`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!");`

```
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
```

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

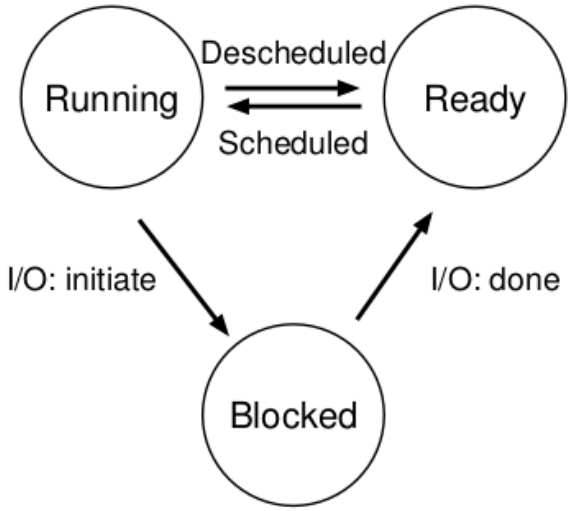


Figure 4.2: **Process: State Transitions**

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.
> V Es verdadero ya que si tenes 10k de procesos con 1 core x ejemplo, entre que se reparten el cpu y que avanza el proceso; el tiempo real sigue avanzando mientras que user+sys avanza muy poquito.

(b) Dos procesos no pueden usar la misma dirección de memoria virtual.
> F. Ambos pueden usar la misma dirección de memoria virtual. Pero que es mapeada a distintas ubicaciones en la memoria física. (Los procesos no se dan cuenta)

(c) Para guardar el estado del proceso es necesario salvar el valor de todos los registros del micropro-
cesador.
>
V (y mas) tambien multiplexo en espacio la ram, stack/heap.

(d) Un proceso puede ejecutar cualquier instrucción de la ISA.
> F
Instrucciones privilegiadas (del kernel) no puedo ejecutarlas desde user space

(e) Puede haber traps por timer sin que esto implique cambiar de contexto.
> Yo diria V
En general tengo 198239813298 interrupts por timer pero puedo mantener el proceso (cuando hay uno x ejemplo)

(f) **`fork()`** devuelve 0 para el hijo, porque ningún proceso tiene PID 0.
> Tiene sentido, V? Init es 1

(g) Las syscall **`fork()`** y **`execv()`** están separadas para poder redireccionar los descriptores de archivo.
> V Si, pescado, para eso se separaron.

(h) Si un proceso padre llama a **`exit()`** el proceso hijo termina su ejecución de manera inmediata.
> F No, el proceso hijo, queda colgando de init.

(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.
> Hay comunicacion padre -> hijo -> padre
comm padre->hijo se hace con argv y argc
comm hijo->padre se hacer con el return y el waitpid()

(j) Nunca se ejecuta el código que está después de **`execv()`**.
> F CONTRAEJEMPLO: Cuando hay error en el execv(), si se ejecuta el código 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()`**.
>
Verdadero, y esos son los procesos Z (zombi), donde el padre se queda esperando el estado de salida del hijo.

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.mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm mmmm

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 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				C	C	C	A	A	A	A	C,A	C	C	C	C,A	A	B,A	A,C	C	C	C					

(I/O)																									
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**	CBA	DCB**A**	DC**B**	DCB	EDC**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.

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

Tturnaround .

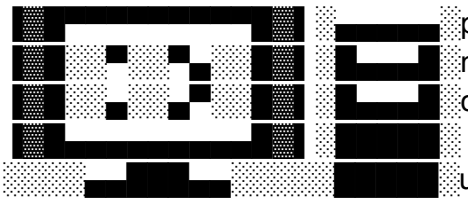
c) La política RR con quanto = ∞ es FCFS.

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

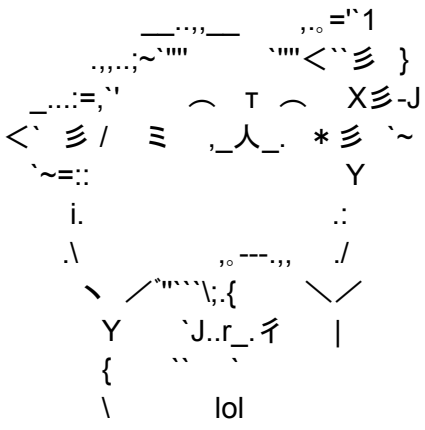
trampas como yield() un poquitito antes del quantum.

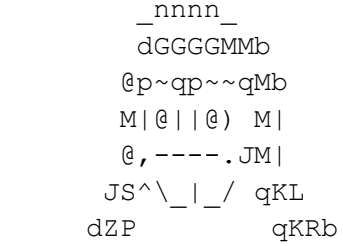
Graffiti zone

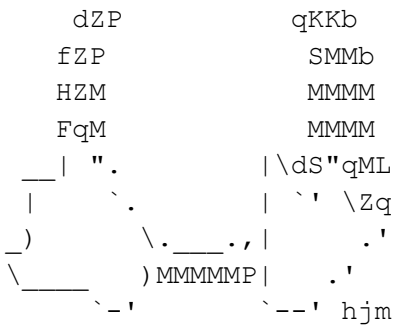


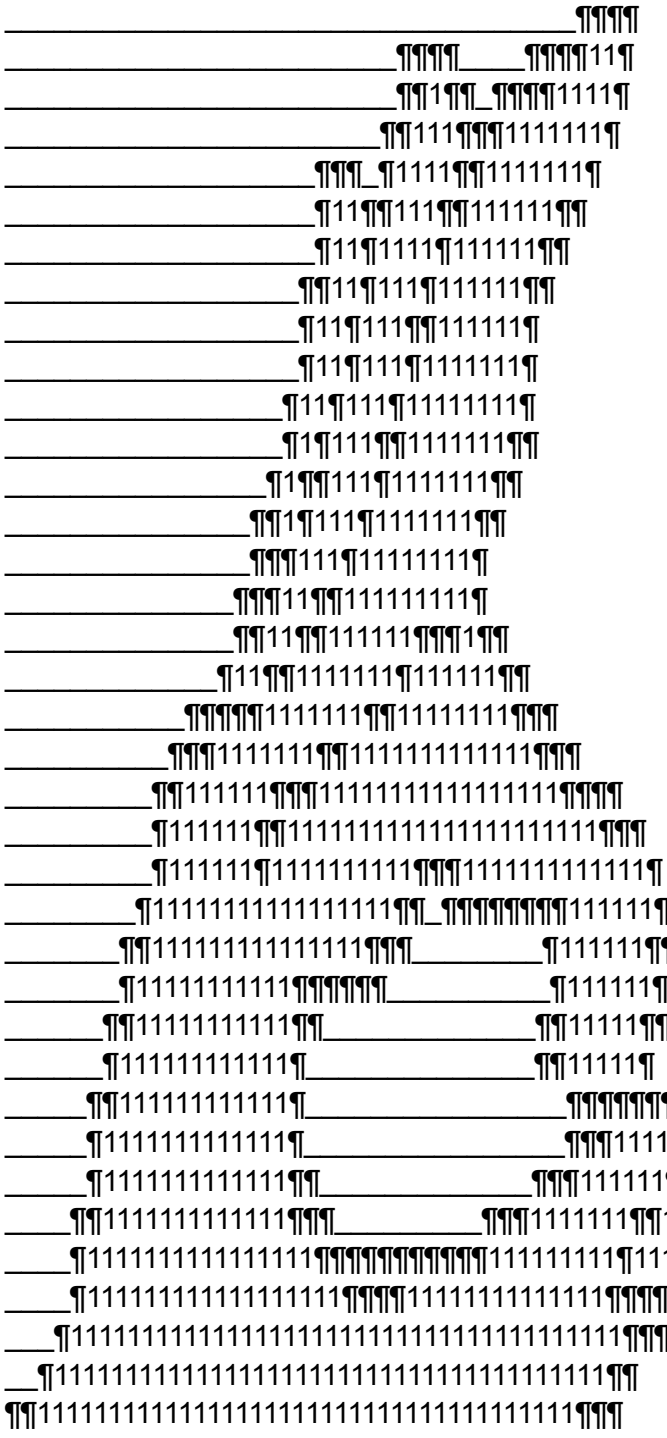
perdon te lo re robe
n.n
chora

uwun’t






dZP qKKb
fZP SMMb
HZM MMMM
FqM MMMM
__| ". |\dS"qML
| `. | `' \Zq
_) \.___.,| .'
____)MMMMMP| .'
`-' `--' hjm



1111111111111111111111111111111¶¶¶¶¶¶¶¶¶
111111111111111111111111111111¶¶¶¶¶
11111111111111111111111111111¶¶¶¶
1111111111111111111111111111¶¶¶
111111111111111111111111111¶¶¶
11111111111111111111111111¶¶¶
1111111111111111111111111¶¶¶
111111111111111111111111¶¶¶
11111111¶¶¶¶¶¶¶¶¶¶¶¶¶
111111¶¶¶¶
¶¶¶¶¶¶¶¶