

Estructura de Datos y Algoritmos EDDA

Órdenes de complejidad:

O: representa el peor caso posible (cota superior)

Omega Ω : representa el mejor caso posible (cota inferior)

Theta Θ : impone una cota superior e inferior sobre el rendimiento de una función, a modo de promedio.

f(n): n representa el tamaño de algún elemento del problema a resolver mediante el algoritmo y f(n) la ejecución del algoritmo mismo.

Para que una función f(n) se considere O(f(n)) o $\Omega(f(n))$ se considere cota superior/cota inferior de una curva, debe ser siempre mayor/menor que está, partiendo desde n_0 .

$$\forall n \geq n_0, c f(n) \geq d g$$

Reglas

1. Se ignoran las constantes

$$3 * O(1) \Rightarrow O(1)$$

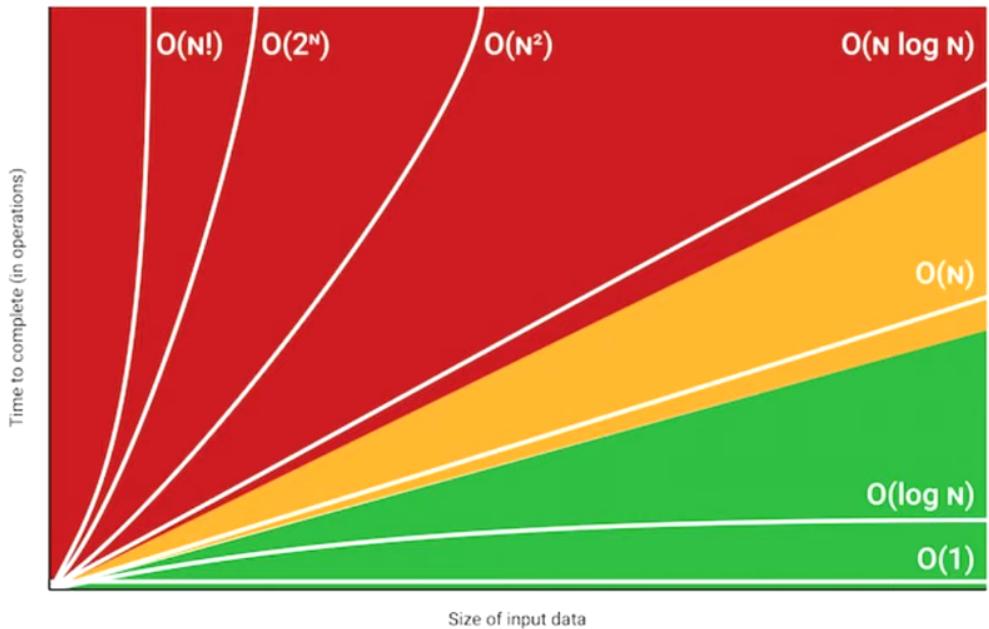
$$O(n/2) = > O(n)$$

2. Dominancia de términos

$$O(n) < O(n^2)$$

$$O(n \log n) < O(2^n)$$

* Nos quedamos con el peor caso (el más complejo)



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$

Cualquier línea de código que no sea un ciclo, una recursión o un llamado a una función que a su vez es alguna de las dos primeras, se considera $O(1)$.

Cuando dos o más procesos se acoplan/anidan, la complejidad aumenta según la cantidad de veces que se repite el ciclo más interno (se puede expresar como que se “multiplican” los O). Ejemplo, un for que recorre un arreglo e imprime cada valor tiene complejidad $O(n)$, mientras que un for anidado dentro de otro (para recorrer una matriz, por ejemplo) que de igual manera imprime cada valor, tiene una complejidad de $n*n*O(1) = O(n^2)$.

```
// Donde "n" es la entrada
for (int i = 1; i <=n; i *= c) { // O(log[n])
    // Cualquier sentencia O(1)
}
for (int i = n; i > 0; i /= c) { // O(log[n])
    // Cualquier sentencia O(1)
}
```

ciclo cuya variable incrementa multiplicándose/dividiéndose.

```
// Donde "n" es la entrada
for (int i = 2; i <=n; i = pow(i, c)) { // O(log[log[n]])
    // Cualquier sentencia O(1)
}
for (int i = n; i > 1; i = fun(i)) { // O(log[log[n]])
    // Cualquier sentencia O(1)
}
```

ciclo cuya variable incrementa de forma exponencial.

Big O para algoritmos recursivos: el tiempo que tarda una ejecución de n ($T(n)$) es igual al tiempo de ejecución anterior + tiempo de ejecución de lo que se hace adicionalmente en el llamado recursivo.

Ejemplo 1:

```
1 def factorial(n):
2     if n == 1:
3         return 1
4     return n * factorial(n-1)
```

$$\begin{aligned}T(n) &\equiv T(n-1) + O(1) \\T(n) &\equiv O(n)\end{aligned}$$

Ejemplo 2:

```

1 def MaxArray(arr):
2     if len(arr) == 1:
3         return arr[0]
4
5     mitad = len(arr)//2
6     arr_izq = arr[0:mitad]
7     arr_der = arr[mitad:]
8
9     max_izq = MaxArray(arr_izq)
10    max_der = MaxArray(arr_der)
11    max_mitad = CalcMaxMitad(arr, mitad)
12
13    return max(max_izq, max_der, max_mitad)

```



► Big O para algoritmos Recursivos | Análisis de Algoritmos

Estructuras fundamentales

Lista:

Ligada: tiene puntero hacia el siguiente dato y el siguiente tiene puntero hacia el anterior (dblemente ligada)

Indexada: Un arreglo, asigna índices a los elementos en cada posición

Pila: LIFO

Cola: FIFO

¿Como hago una cola y una pila como lista ligada? ¿Cómo sería borrar y agregar un valor en ellas?

R: Si interpretamos una cola como lista ligada, se agregaría el elemento al final sin problemas, haciendo que apunte al último, mientras que para sacar, simplemente se copia y se borra (nadie apuntaba a él). Por otra parte, para una pila, se agrega de igual manera pero para borrar hay que anular la referencia del elemento anterior que apuntaba a él.

Algoritmos de ordenamiento + costos de complejidad y estabilidad

BubbleSort: $O(n^2)$

BubbleSort va ordenando de a pares (dos elementos consecutivos) viendo cual es menor, para luego emparejar al mayor con el siguiente y repitiendo el proceso hasta n veces (con cada iteración se asegura que el último valor es el mayor).

InsertSort: $O(n^2)$

Partiendo por el primer valor, voy comparando si el valor a la izquierda es menor. Si es menor, paso al siguiente. Si el valor a la izquierda es mayor, traslado el valor actual hacia la izquierda casilla por casilla hasta que a su izquierda se encuentre un número menor.

HeapSort: $O(n \log n)$

En HeapSort se crea un min-heap con los elementos de un árbol, se elimina el nodo raíz, se escoge el último valor como nueva raíz y se reconstruye el árbol (heapify) en tiempo logarítmico en cada iteración.

MergeSort: estable en $O(n \log n)$

Se divide un arreglo en mitades hasta obtener solo pares, los cuales se ordenan y luego se vuelven a unir.

QuickSort: $O(n \log n)$ si el pivote es bueno, $O(n^2)$ si el pivote es malo.

Se escoge un pivote, el cual se mueve al final del arreglo. QuickSort busca un ítem mayor que el pivote de izquierda a derecha, y un ítem menor de derecha a izquierda. Una vez encontrados estos dos valores, se intercambian las posiciones. Esto se detiene cuando el ítem menor tiene índice menor al ítem mayor (está ubicado antes). Una vez hecho esto, se devuelve el pivote a su ubicación original intercambiándolo con el ítem mayor. Todo este proceso se va a repetir también para los tramos a la izquierda y derecha del pivote hasta que todo el arreglo está ordenado.

obs: HeapSort y MergeSort suelen ser mejores que QuickSort, por el tema de estabilidad. QuickSort escoge un pivote y ubica los elementos menores a un lado y mayores al otro, por lo que el rendimiento de este algoritmo depende de la correcta elección del pivote, de lo contrario puede incluso llegar a costar n^2 .

BDD

Aridad: cantidad de entidades que se relacionan entre sí bajo la misma relación

La PK no se pone en el diagrama ER.

Cuando hay una aridad mayor a 2 hay que crear una nueva tabla con el nombre de la relación que contenga las PK de las tablas como FK.

En el modelo relacional no pueden aparecer tablas que no correspondan a entidades del modelo E-R, a excepción de las tablas que correspondan a relaciones y de aridad (vínculos).

Ojo con:

- Surgen entonces tres tipos de restricciones de integridad:
 - Restricciones de dominio
 - Restricciones de entidad
 - Restricciones referenciales

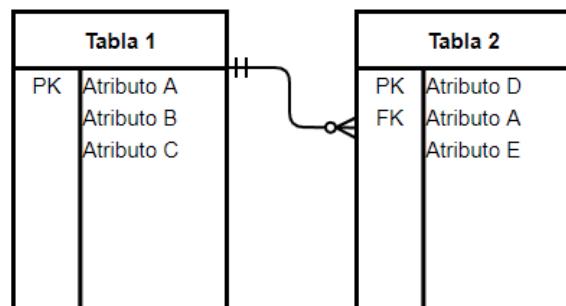
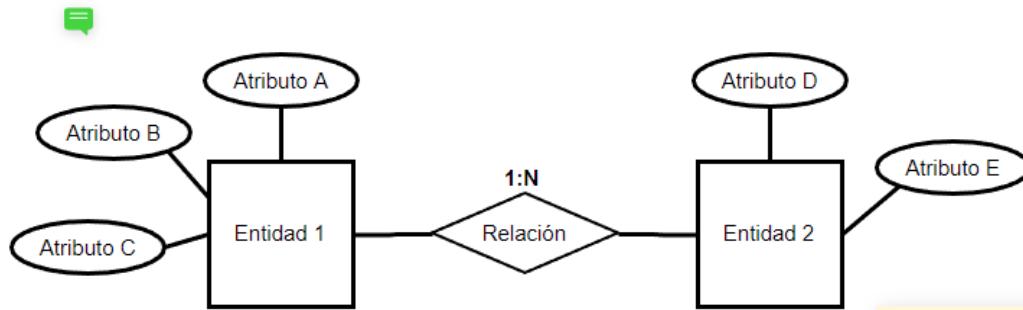
dominio: datos consecuentes; entidad:

Normalización

Mayor normalización agrega redundancia y rigurosidad, a cambio de rendimiento y tamaño.

LEER CAP 3B

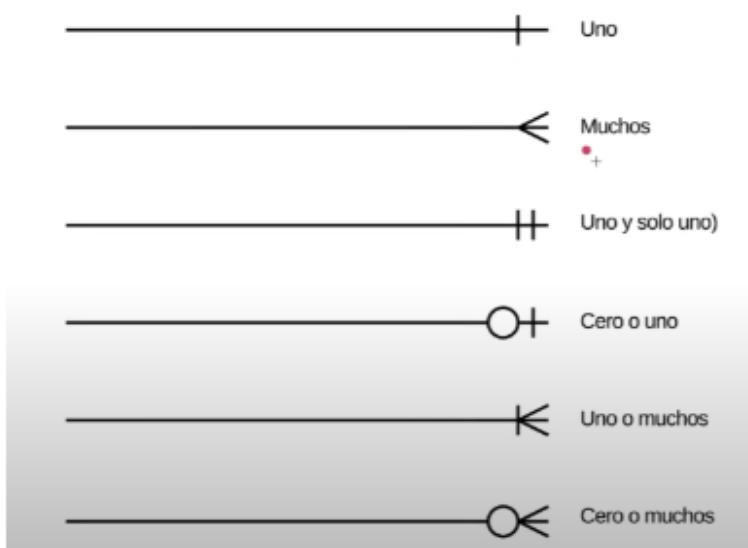
Entidad Relación



E-R a Relacional

- Si es 1:N, los N absorben la PK del 1, por lo que son los que tendrán la FK.
- Si es N:M, se ocupa la relación como tabla intermedia, que contendrá ambas PK como FK.
- Si una tabla N es una entidad débil (sin PK) entonces es absorbida por la tabla de la relación teniendo la PK de la tabla 1 como FK.

Cardinalidad



SQL

SQL I

Funciones de agregación

COUNT: SELECT COUNT(atributo) FROM tabla WHERE
Complementado con DISTINCT crea una cuenta sin repetidos.
Ej: SELECT COUNT(DISTINCT atributo) FROM tabla WHERE...

MAX: SELECT MAX(atributo) FROM tabla WHERE

MIN: SELECT MIN(atributo) FROM tabla WHERE

AVG: SELECT AVG(atributo) FROM tabla WHERE

SUM: SELECT SUM(atributo) FROM tabla WHERE

SQL II

ORDER BY: SELECT sentencia ORDER BY atributo ASC/DESC

GROUP BY: SELECT sentencia GROUP BY atributo

HAVING: SELECT sentencia HAVING condicion (con funciones de agregación)

Ej: Select sentencia WHERE condicion HAVING COUNT(atributo) > 3

SQL III

INSERT: INSERT INTO tabla VALUES (valor1, valor2, valor3) // todos los valores y en el orden de la tabla en la bdd

UPDATE: UPDATE tabla SET atributo1 = "valor", atributo2 = 123 WHERE

DELETE: DELETE FROM tabla WHERE

DROP TABLE: DROP TABLE tabla;

CREATE TABLE: CREATE TABLE tabla (columna1 datatype constraint, columna2 datatype constraint, columna3 datatype constraint....)
// datatypes comunes: int(), varchar(), boolean, date.

//constraints comunes:

NOT NULL, AUTO_INCREMENT, DEFAULT valor

+

primary key: PRIMARY KEY(atributo) al final del query

foreign key: FOREIGN KEY (atributo) REFERENCES otra_tabla(atributo de la otra tabla) al final del query

Operadores Lógicos

Selección => $R' = R \text{ where } \text{condicion}$

Proyección => $R' = R[\text{atributo}]$

Resta => $R' = R1 \text{ SUBTRACT } R2$ // Ocurrencias de R1 que no estén en R2

Producto => $R' = R1 \text{ TIMES } R2$ // Ocurrencias de R1 y R2, es decir, concatena a R1 y a R2

Asociación => $R' = R1 \text{ JOIN } R2 \text{ (condicion)}$

Order by (Relación, atributo) // Utilizar group by cada vez que en el enunciado diga "cada"

Group by (relación, atributo, función)

2. Procesos

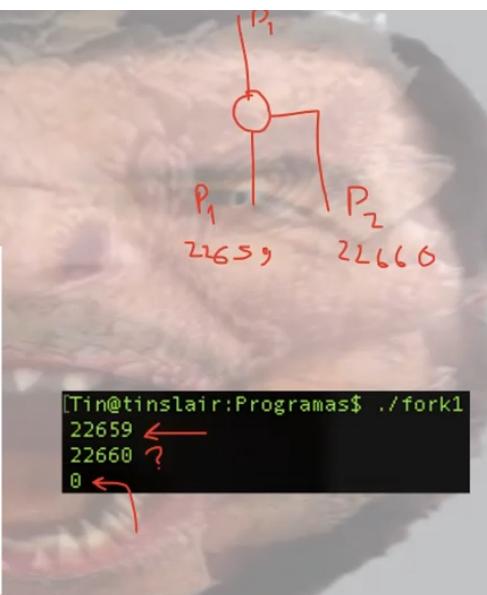
- fork()

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]){

    pid_t me, childpid;
    me = getpid();
    printf("%d\n", me); ←
    childpid = fork();
    printf("%d\n", childpid);

    return 0;
}
```



```
[Tin@tinslair:Programas$ ./fork1
22659 ←
22660 ? ←
0 ←
```

Aquí vemos como el proceso padre (22659) está retornando el pid de su hijo (22660), y a su vez, como es una copia del padre, childpid repite la sección del código que viene después de su creación, imprimiendo un 0 ante el último print, dado que el no posee childpid. Esto sirve para identificar cuál proceso es el padre (posee childpid) y cuál es hijo (childpid = 0).

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]){

    int status;
    pid_t me, childpid;
    me = getpid();
    printf("%d\n", me);
    childpid = fork();

    if(childpid == 0)
    {
        printf("Proceso hijo: Ejecutando.\n");
        printf("Bienvenidos a Temuco.\n");
        exit(9);
    }
    else if(childpid > 0)
    {
        printf("Proceso padre: el PID del proceso hijo es: %d\n", childpid);
        wait(&status);
    }

    printf("Sugar daddy: El proceso hijo termino.\n");
    printf("Status de salida: %d\n", WEXITSTATUS(status));

    return 0;
}
```

```
[Tin@tinslair:Programas$ ./exit1
23145
Proceso padre: el PID del proceso hijo es: 23146
Proceso hijo: Ejecutando.
Bienvenidos a Temuco.
Sugar daddy: El proceso hijo termino.
Status de salida: 9
```

El último print del hijo no se ejecuta porque el exec lo hace cambiar de código. El padre espera que haya un status de salida (en el hijo), y al final lo imprime.

- kill()
- sleep()
- Buscar y probar killall()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main(int argc, char *argv[]){
    pid_t childpid;
    childpid = fork();

    if(childpid > 0)
    {
        int i = 0;
        while(i++ < 5)
        {
            printf("Proceso padre...\n");
            sleep(1);
        }
        kill(childpid, SIGKILL);
    }
    else if (childpid == 0)
    {
        int i = 0;
        while(i++ < 15)
        {
            printf("Proceso hijo...\n");
            sleep(1);
        }
    }
    else
    {
        printf("Algo fallo...\n");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

```
[Tin@tinslair:Programas$ ./kill1
Proceso padre...
Proceso hijo...
Proceso hijo...
Proceso padre...
Proceso padre...
Proceso hijo...
Proceso hijo...
Proceso hijo...
Proceso padre...
Proceso hijo...
Proceso hijo...
Proceso hijo...
Proceso hijo...
```

Obs: Si un proceso hijo sigue ejecutándose cuando el padre ya se cerró, este hijo se vuelve huérfano.

Obs2: Si el proceso hijo se cierra y el proceso padre no llama a wait o waitpid para obtener la información de estado del hijo, el pedido del proceso hijo aún se almacena en el sistema, considerándose un proceso zombie.

Obs3: El control de procesos tiene que hacerse en C.

Scheduling

- Long term scheduler:
 - Se refiere a la selección de los procesos que quedan en la Ready Queue y define su cantidad.
- Medium term scheduler
 - Modifica temporalmente el grado de multiprogramación y hace swapping (RAM → HDD y HDD → RAM)
- Short term scheduler (dispatcher)
 - Manda procesos de la Ready Queue a CPU y efectúa el cambio de contexto.

Trashing: gasto demasiado tiempo y recursos gestionando para lo poco que proceso.

- Scheduling expropiativo
- Scheduling no-expropiativo
- Batch scheduling
- Interactive scheduling
- Real-time scheduling

En todos los casos, se debe buscar que todo proceso tenga oportunidad de utilizar la CPU durante un periodo razonable.

expropiativo: si hay un proceso ejecutándose, puede detenerse.

no-expropiativo: lo contrario.

batch: yo entrego una serie de operaciones al scheduler

interactivo: va ocurriendo con cierta interacción del usuario (dispositivos de I/O)

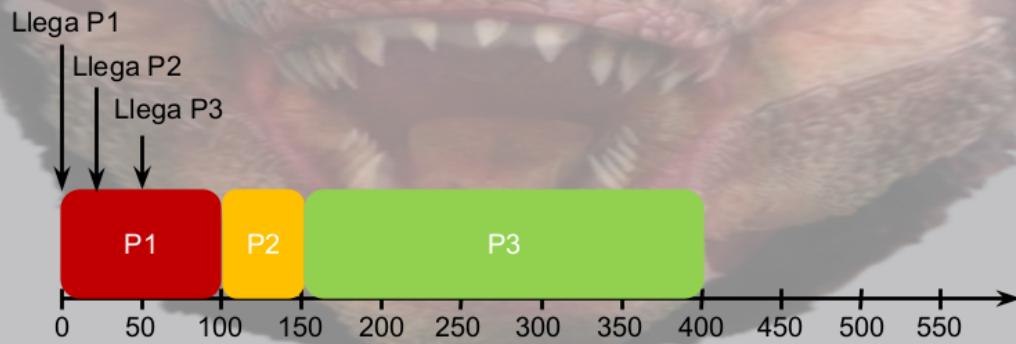
tiempo-real: es en tiempo real XD.

- A continuación se describen algunos algoritmos de scheduling de los tipos mencionados:
 - First come first served
 - Shortest job first
 - Round-robin
 - Priority
 - Multi-level Feedback Queue
 - Earliest deadline first
- } Batch } Interactive } Real-time

2. Procesos

- First come first served

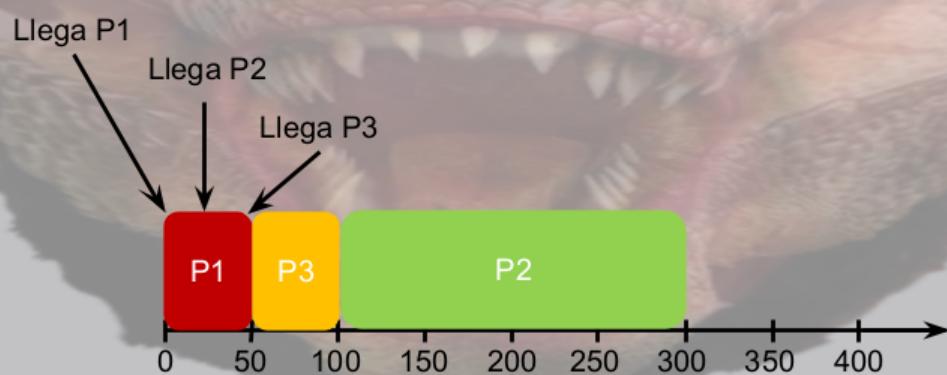
Process	Arrival time	CPU burst	End time	Turnaround
P1	0	100	100	100
P2	20	50	150	130
P3	50	250	400	350



2. Procesos

- Shortest job first

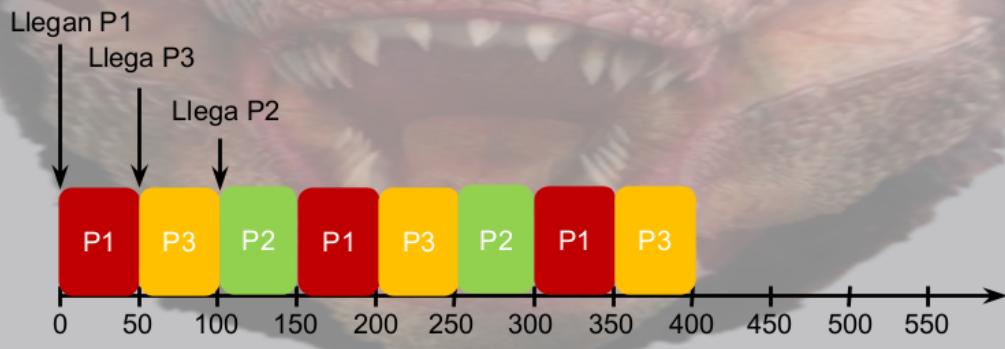
Process	Arrival Time	CPU Burst	End Time	Turnaround
P1	0	50	50	50
P2	25	200	300	275
P3	50	50	100	50



2. Procesos

- Round-robin (q=50)

Process	Arrival time	CPU burst	End time	Turnaround	Execution start	Response time
P1	0	150	350	350	0	0
P2	100	100	300	200	100	0
P3	50	150	400	350	50	0

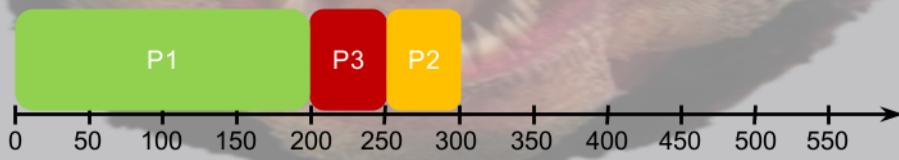


2. Procesos

- Priority scheduling

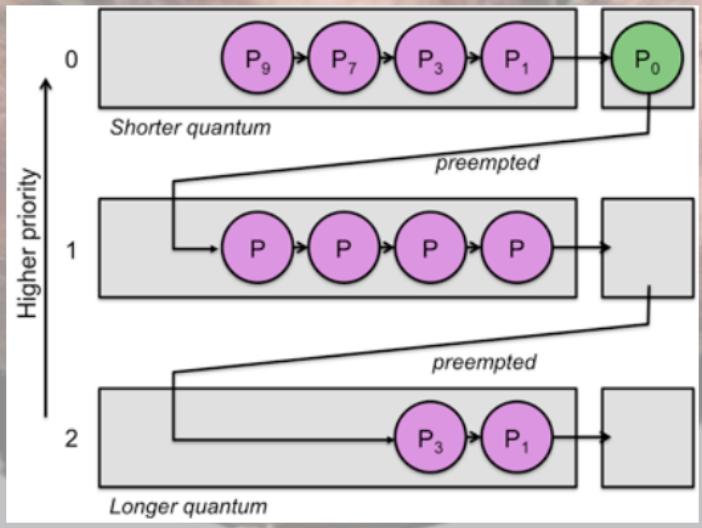
Process	Arrival time	CPU burst	End time	Turnaround	Execution start	Response time	Priority
P1	0	200	200	200	0	0	63
P2	0.1	50	300	299.1	250	249.9	24
P3	0.2	50	250	249.8	200	199.8	50

Ver: http://osr507doc.xinuos.com/en/PERFORM/calc_proc_priorities.html



2. Procesos

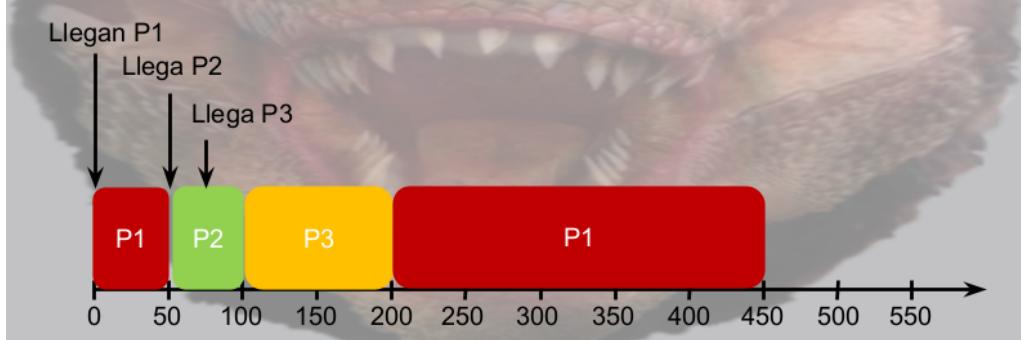
- Multilevel Feedback Queue (MLFQ)



2. Procesos

- Earliest deadline first

Process	Arrival time	CPU burst	End time	Turnaround	Execution start	Response time	Deadline
P1	0	300	450	450	0	0	500
P2	50	50	100	50	50	0	350
P3	75	100	200	125	100	25	450



Herramientas de Sincronización:

Locks de MUTEX: deja pasar al primero que toma el lock y se bloquea (garantiza atomicidad de su manejo y exclusión mutua). Usa las primitivas Acquire y Release.

Semáforos: es una versión de lock para multi-threads. Se implementa mediante un contador que se incrementa/decrementa atómicamente.

- **Semáforos**

- Para un semáforo S, se implementan dos funciones:

- P(), wait() → intenta decrementar el valor del contador para simbolizar que ha entrado un thread en la SC.
 - V(), signal() → intenta incrementar el valor del contador. Representa una ubicación libre más para que otro thread pueda entrar a la SC.

- Cuando el contador del semáforo llega a 0, no pueden entrar más threads a la SC.

```
sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntro a la SC\n");

    //critical section
    printf("\n Como soy tan bacan... me voy a dormir durante la SC... \n");
    sleep(4);
    printf("\n Buenos dias!!! \n");

    //signal
    printf("\nSalgo de la SC\n");
    sem_post(&mutex);
}

int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

Implementación de sincronización de threads usando semáforos

Variables de condición: mecanismos de sincronización en que la condición no viene de limitar la cantidad de threads que acceden a la SC, sino que se trata de condiciones arbitrarias. Trabaja con wait() (que siempre bloquea el thread) y signal() (que despierta a un thread bloqueado en caso que lo haya). -> Es como un mutex más complejo, con condicionales (como if).

Monitor: se trata de un objeto que contiene funcionalidades y es resguardado por un lock en su totalidad al ser accedido por un thread. Cualquier operación del monitor queda bloqueada hasta que el thread que llama libera el lock. -> Ocupa las herramientas de sincronización anterior mencionadas para aplicar sincronización sobre un objeto particular (abstracción).

Problemas de acceso

Deadlocks: situación de bloqueo del sistema, donde dos o más threads se quedan bloqueados para siempre, esperandose entre ellos.

- **Para manejar deadlocks, se tienen las siguientes estrategias:**
 - Prevención/evitamiento de deadlock (no dejar que el sistema llegue a ese estado)
 - Detección y recuperación (dejar que ocurra y luego aplicar resolución y posteriormente prevención para evitar que ocurra)
 - Ignorar el problema (sorprendentemente, es la estrategia empleada por Windows y UNIX)

- Dichos deadlocks se pueden dar en presencia de las siguientes condiciones (necesarias y simultáneas):
 - Exclusión mutua
 - Captura y espera
 - Espera circular
 - No expropiación
- Surgen del incumplimiento de las propiedades de SC. Representan la negación de la propiedad de progreso.

Livelocks: situación de bloqueo del sistema, donde dos o más threads cambian de estado constantemente pero aun así no progresan.

- El livelock es un caso especial de inanición y también incumplen la propiedad de progreso.
- Frecuentemente, los livelocks son resultado de un intento de manejo de deadlock.

