**Formule contigua, linkata e indicizzata:**

#blocchi = size disco/size blocco;

size puntatore = log2 #blocchi;

blocco del byte = byte da cercare / size blocco;

Nella linkata fare = byte da cercare / (size blocco – size puntatore);

**SOLO INDICIZZATA**

#puntatori in un blocco= size blocco/ size puntatore (per difetto);

(per l’indicizzata concatenata togliere il puntatore finale usato per puntare al prossimo blocco indice)

#blocchi indice = numero blocchi/ #puntatori in un blocco (per eccesso);

**Lista Linkata:**

blocchi occupati= #byte file / size blocco senza il puntatore;

blocchi liberi= #blocchi - blocchi occupati;

**BITMAP:**

#blocchi= size disco/size blocco;

size bitmap= #blocchi;

**FILE e RECORD:**

size File blocco indice = #puntatore in un blocco \* size blocco;

#record in un blocco= size blocco / size record;

#blocchi file con record= size file / #record in un blocco;

**FAT:**

#blocchi FAT= 2 elevato (bit della size puntatore);

size FAT= #blocchi \* size puntatore;

massima capacità= #blocchi \* size blocco;

**Pagine e frame:**

Dimensione entry table= #pagine;

Dimensione RAM= #frame;

Offset= logaritmo in base 2 della size pagine o della size frame;

Numeri bit pagina= logaritmo in base 2 del #pagine;

Numeri bit frame= logaritmo in base 2 del #frame;

Memoria Virtuale=Si ha quando l’indirizzo logico ha più bit dell’indirizzo fisico ed ha obbligatoriamente il bit di validità;

dim page table= bit frame+(bit validità, bit modifica, bit di riferimento);

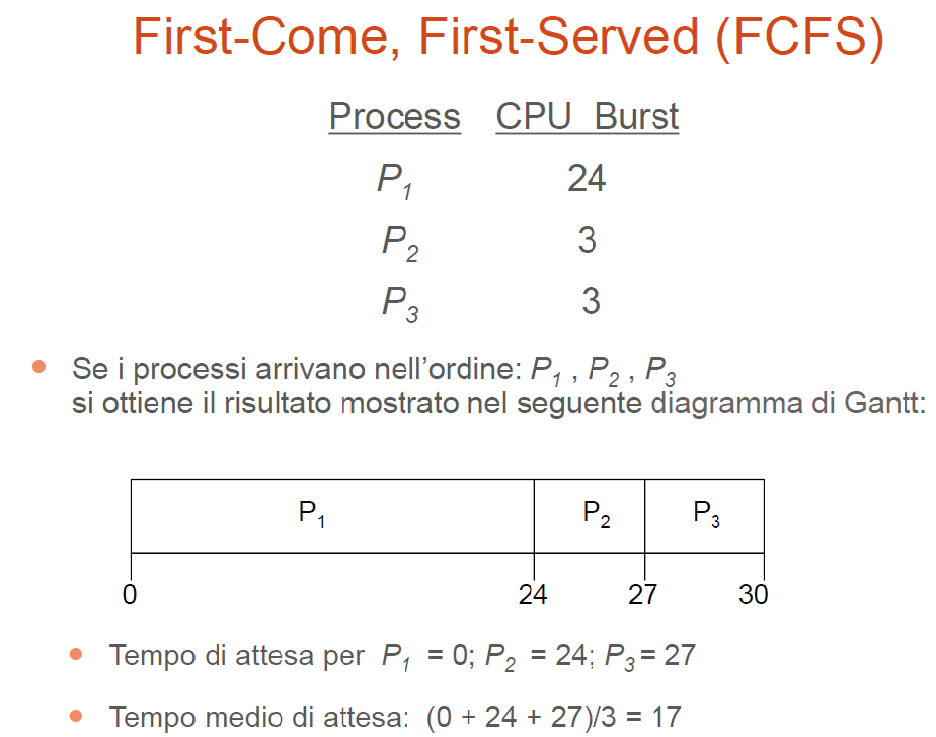
max page table = [bit frame + (bit validità, bit modifica, bit di riferimento)] \*2elevato(bit page);

***Frammentazione esterna***, quando c’è abbastanza spazio di memoria centrale per soddisfare una richiesta, ma gli spazi disponibili non sono contigui, può portare ad un blocco libero.

**Frammentazione interna**, quando si alloca la memoria in blocchi di dimensione variabile e si caricano/rimuovono da quest’ultima dei processi, lo spazio libero si frammenta in piccole parti.

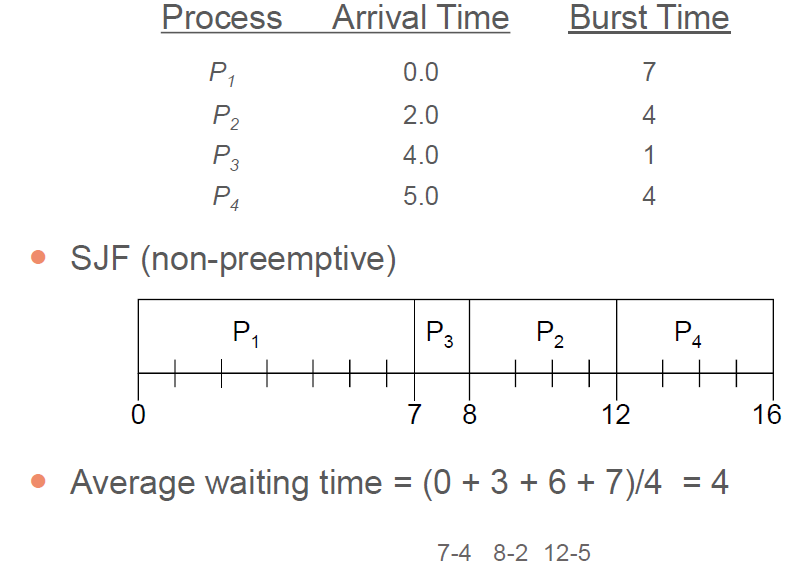




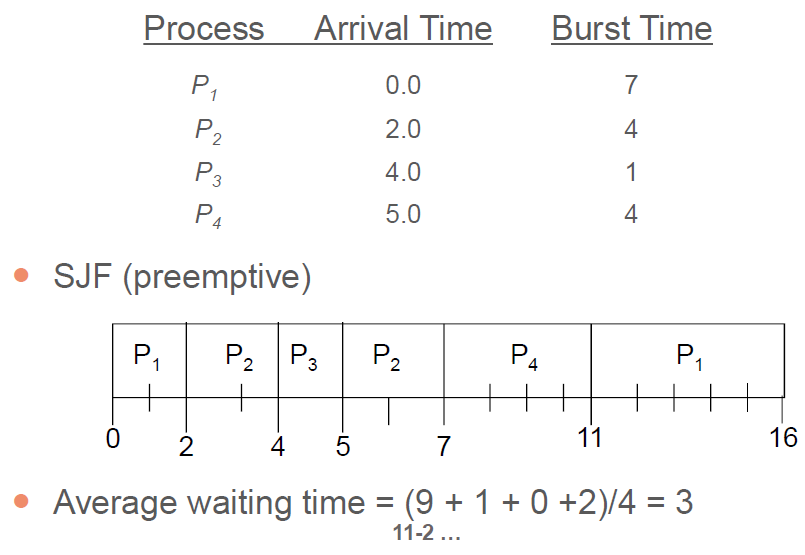


Shortest-Job-First (SJF)

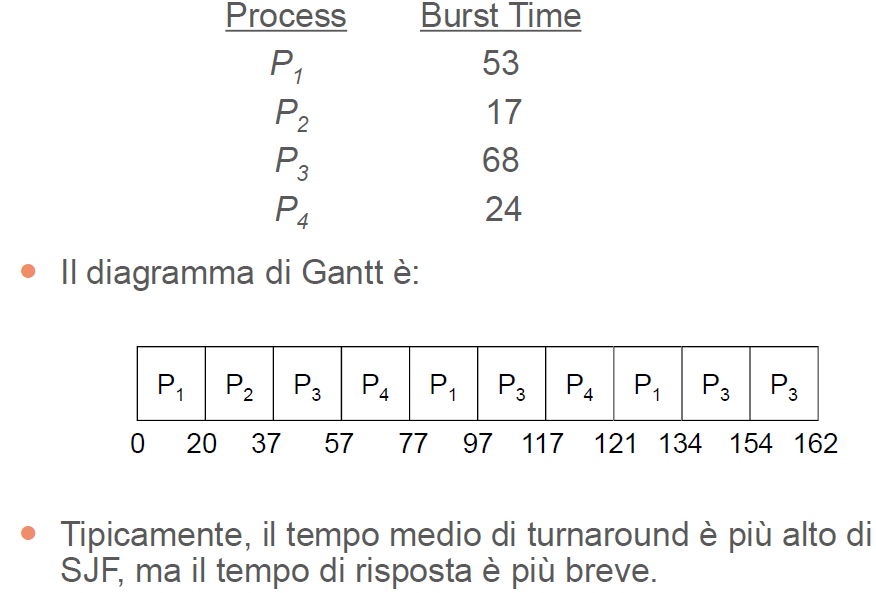
***Nonpreemptive***, quando un processo ha ottenuto la CPU fino al completamento del suo cpu-burst.



***Preemptive***, quando un nuovo processo è pronto, ed il suo CPU-burst è minore del tempo di cui necessita ancora il processo in esecuzione, c’è prelazione.



Round Robin (RR)

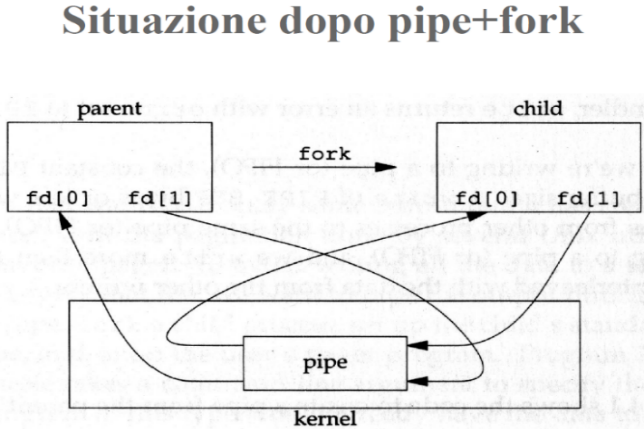
Ogni processo riceve la CPU per una piccola unità di tempo

int ***pipe***(int filedes[2] );

• filedes[0] è il file descriptor di un "file" aperto in lettura;

• filedes[1] è il file descriptor di un "file" aperto in scrittura;

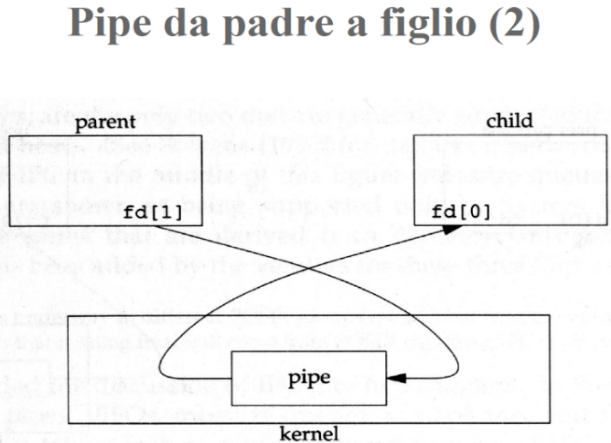
• inoltre l'output di filedes[1] corrisponde all'input di filedes[0].

int fd[2];

…

pipe(fd);

pid=fork();



if(pid>0) { // padre

close(fd[0]);}

else if(pid==0) { // figlio

close(fd[1]);}

Funzione ***write***;

Quando la pipe si riempie (la costante PIPE\_BUF specifica la dimensione), la write si blocca fino a che la read non ha rimosso un numero sufficiente di dati.

Funzione ***read;***

Se la pipe è vuota la read si blocca fino a che non vi siano dei dati disponibili.

Funzione ***close:***

La funzione ***close*** sul descrittore del file in scrittura agisce come *end-of-file* per la ***read***.

La chiusura del descrittore del file in lettura causa un errore nella ***write***.

Int fd[2], fd1[2];

pid\_t pid;

char nomefile[40], carattere[40];

int contatore = 0;

pipe(fd);

pipe(fd1);

pid=fork();

if(pid>0){

close(fd[0]);

close(fd1[1]);

printf(“Inserire nome file: ”);

scanf(“%s”, nomefile);

write(fd[1],nomefile,40);

read(fd1[0], ***&***contatore, sizeof(int));

} else if(pid==0){

close(fd[1]);

close(fd1[0]);

read(fd[0], nomefile, 40);

FILE\* in=fopen(nomefile, ”r+”);

while( fscanf(in, ”%c”, carattere) != EOF){

if(strcmp(carattere,” ”) == 0) contatore++;

}

write(fd1[1], ***&***contenuto, sizeof(int));

} }