GENERAZIONE PROCESSO FIGLIO
(padre attende terminazione del figlio)

```
#include <stdio.h>
void main (int argc, char *argv[])
{
        pid = fork(); /* genera nuovo processo */
        if (pid < 0) { /* errore */
        fprintf(stderr,“creazione nuovo processo fallita”);
        exit(-1);
}
        else if (pid == 0) { /* processo figlio */
        execlp(“/bin/ls”, “ls”, NULL);
}
        else { /* processo padre */
        wait( NULL); /* attende terminazione figlio */
        printf(“il processo figlio ha terminato”);
        exit(0);
}
}
```

GENERAZIONE THREAD
(tramite API pthreads)

```
#include <pthread.h>
#include <stdio.h>
int somma;                              /* questo dato è condiviso dai thread */
void *runner(void *param)               /* il thread */
void main (int argc, char *argv[])
{
        pthread_t tid;                  /* identificatore del thread */
        pthread_attr_t attr;            /* attributi del thread */
        if (arcgc != 2) || (atoi(argv[1]) < 0) /* errore */
                exit(-1);
        else {
                                        /* reperisce gli attributi predefiniti */
        pthread_attr_init(&attr);
                                        /* creazione del thread */
        pthread_create(&tid,&attr,runner,argv[1]);
                                        /* attende la terminazione del thread */
        pthread_join(tid,NULL);
}
}  [CONTINUA]
```

```
                                        /* codice del thread */
void *runner(void *param)
{
        int sup, i;
        sup = atoi(param);
        somma = 0;
        if (sup > 0) {
        for(i=1;i<=sup;i++)
        somma += 1;
}
pthread_exit(0);            /* terminazione del thread */
}
```

SCHEDULING DELLA CPU

FIRST COME FIRST SERVED (FIFO)

→I processsi vengono messi a running in ordine di arrivo e tengono la CPU fino alla fine del loro CPU burst senza interruzioni

ALGORITMO PER PRIORITA’

→ La CPU viene allocata al processo con la priorità più alta (numero piccolo ≡ alta priorità).
• Con prelazione
• Senza prelazione

→ Problema ≡ Starvation – processi con bassi priorità potrebbero non ricevere mai la CPU.
→Soluzione ≡ Aging – con il passare del tempo nella coda di ready, la priorità viene aumentata.

SCHEDULING CPU – SHORTEST JOB FIRST

→ La CPU viene assegnata al processo con il CPU burst successivo più breve.

→Due schemi:
• Senza prelazione – il processo mantiene il possesso della CPU fino a quando non completa il CPU burst attuale.
• Con prelazione – se arriva un nuovo processo con un CPU burst inferiore del CPU burst che resta al processo attualmente in esecuzione. Questo schema viene chiamato anche Shortest-Remaining-Time-First (SRTF).

→ SJF è ottimale – minimizza il tempo medio di attesa.

STIMA DEL PROSSIMO CPU BURST

- $\mathbf{t_n}$ = lunghezza effettiva dell'n-esimo CPU burst
- $\boldsymbol{\tau_n}$ = lunghezza stimata dell'n-esimo CPU burst
- $\boldsymbol{\alpha}$ = peso, dove $0 \leq \alpha \leq 1$
- **c** = stima iniziale del primo CPU burst

$$\boldsymbol{\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n}$$

$$\boldsymbol{\tau_0 = c}$$

ROUND ROBIN

→ Ogni processo riceve la CPU per una piccola unità di tempo (quanto di tempo), di solito 10-100 millisecondi. Allo scadere di questo quanto di tempo il processo è prelazionato ed aggiunto alla coda dei processi ready.

→ La coda dei processi di ready viene gestita come una coda (FIFO) circolare.

→ Se ci sono n processi nella coda di ready e il quanto di tempo è q, allora ogni processo non aspetta più di (n-1)q unità di tempo.

→Le prestazioni dipendono da q
• q molto grande => RR diventa FCFS
• q molto piccolo => q deve essere molto più grande del tempo di commutazione di contesto, altrimenti l'overhead è troppo elevato

SCHEDULING A CODE MULTIPLE

Questo algoritmo di scheduling suddivide la coda di ready indiverse code distinte.

Ogni coda adotta un proprio algoritmo di schedulinoÈ inoltre necessario avere uno scheduling tra le code.

Un processo può passare da una coda ad un'altra.
→Ad esempio un processo che usa troppo la CPU si può spostare in una coda con priorità minore.
→ Questo schema mantiene i processi con prevalenza di I/O e i processi interattivi nelle code con priorità elevata.
→ Analogamente un processo che attende da troppo tempo si può spostare in una coda con priorità maggiore (ovvero questo è un modo per implementare l'aging).

SCHEDULING CPU – LINUX

→ Linux usa due algoritmi di scheduling:
• time-sharing quando è più importante l'equa distribuzione del tempo
• real-time quando le priorità sono più importanti dell'equità

→L'identità di un task contiene anche la classe di scheduling, attraverso la quale si determina quale algoritmo applicare. Tali classi sono:
• time-sharing
• real-time RR
• real-time FCFS:

→ Un task in classe real-time ha priorità statica e goodness più alte rispetto ad un task in classe time-sharing

SCHEDULING TIME-SHARING

→ algoritmo per priorità basato sui crediti (RR con quanto dinamico).

• Ogni task ha:
- rt_priorità = 0
- $ts\_priorità_0$ = 20 – nice dove $-19 \leq nice \leq 20$
- $crediti_0$ = priorità / 4 (di fatto è il suo quanto di tempo)
- $goodness_n = ts\_priorità_n + crediti_n$

• tra i task time-sharing la scelta cade sul task che ha il maggior numero di crediti.
• Ogni volta che si ha una interruzione dal timer, il task in esecuzione perde un credito.
• Quando i suoi crediti arrivano a zero, viene scelto un altro task.
• Quando tutti i task hanno credito zero, vengono ricalcolati i crediti di tutti i task del sistema (non solo quelli ready) con la formula:

$$\mathbf{crediti_{n+1} = (crediti_n / 2) + (ts\_priorità_n / 4)}$$

SCHEDULING REAL TIME

→ È uno scheduling real-time debole

• Ogni task ha:
- $1 \leq$ rt_priorità ≤ 99
- $ts\_priorità_0$ non definita
- $crediti_0$ non definita
- goodness = rt_priorità + 1000

• Lo scheduler esegue i task con la rt_priorità più alta, a parità di rt_priorità quelli che attendono da più tempo.

→ I task in classe FIFO continuano fino a quando non rilasciano la CPU spontaneamente o arriva un task con rt_priorità maggiore.
→ I task in classe RR vengono interrotti allo scadere del loro quanto di tempo e rimessi in coda oppure quando arriva un task con rt_priorità maggiore.

ALLOCAZIONE DELLA MEMORIA
(ALLOCAZIONE CONTIGUA A PARTIZIONI VARIABILI)

→First-fit: Alloca il primo buco di dimensioni sufficienti.
→Best-fit: Alloca il buco più piccolo in grado di soddisfare la richiesta; bisogna scandire l'intera lista (a meno che non sia ordinata per dimensione). Produce le parti di buco inutilizzate più piccole.
→Worst-fit: Alloca il buco più grande; bisogna scandire l'intera lista (a meno che non sia ordinata per dimensione). Produce le parti di buco inutilizzate più grandi.

First-fit e best-fit sono migliori di worst-fit in termini di velocità e utilizzo della memoria.

ALLOCAZIONE DELLA MEMORIA – PAGINAZIONE

→ Lo spazio degli indirizzi logici può non essere contiguo.
→La memoria fisica è divisa in blocchi di dimensioni fisse chiamati frame (le dimensioni sono potenze di 2, tra 512 byte e 8192 byte).
→ La memoria logica è divisa in blocchi chiamati pagine, le cui dimensioni sono pari a quelle dei frame.

→ Per eseguire un programma di dimensioni pari ad n pagine, bisogna trovare n frame liberi e caricare le pagine in quei frame.
→ Viene usata una tabella delle pagine (di solito una per processo) per tradurre gli indirizzi logici in indirizzi fisici.

TEMPO DI ACCESSO ALLA MEMORIA

1) PAGINAZIONE

(accesso in RAM) $T_{RAM}=1$ → $EAT_{medio} = 1 + 1 = 2$ (2 accessi in ram)
(accesso in cache) $T_{TLB}=\varepsilon < 1$ → $EAT_{medio}= \varepsilon + 2 + \alpha$ (α= prob. elemento in cache)

→Metter parte della tabella delle pagine nella memoria associativa (cache) conviene solo se $\varepsilon < \alpha$

2) PAGINAZIONE SU RICHIESTA (memoria virtuale)

Probabilità che si verifichi un Page Fault: $0 \leq p \leq 1.0$

- p = 0 page fault non si verifica mai
- p = 1 page fault si verifica sempre

EAT = (1 – p) * tempo_accesso_in_memoria + p (page_fault + swap_out + swap_in+ restart)

ALGORITMI DI SOSTITUZIONE DELLE PAGINE

FIFO

Sostituire la prima pagina caricata in memoria

OTTIMO

Sostituire la pagina che si utilizzerà il più tardi possibile.

LRU

Sostituire la pagina che è stata usata meno di recente

→ **Con i Contatori**

• Ogni volta che c'è un rifermento alla pagina, il valore del clock viene copiato nel contatore.

• si sceglie la pagina che ha il valore più piccolo del contatore.

→ **Con lo Stack**

• I numeri delle pagine sono organizzati in una lista doppiamente concatenata.

• Ogni volta che c'è un riferimento ad una pagina, il suo numero viene inserito/spostato in cima allo stack.

• Quando bisogna selezionare una pagina, si sceglie quella presente in fondo allo stack => la sostituzione non richiede nessuna ricerca nella lista

BIT DI RIFERIMENTO

• bit di riferimento, inizialmente = 0
• Ogni volta che c'è un accesso alla pagina il bit viene posto ad 1.
• La lista delle pagine è gestita FIFO, viene sostituita la pagina che ha il bit di riferimento a 0 (se esiste).

SECONDA CHANCE

• La lista delle pagine è una lista circolare (per questo viene detto anche algoritmo dell'orologio) gestita FIFO, viene selezionata una pagina:
• Se il bit di riferimento è a 0 => viene sostituita
• Se il bit di riferimento è a 1 => il bit viene messo a 0,
- la pagina è lasciata in memoria,
- viene selezionata la pagina successiva a cui vengono applicate le stesse regole

ASSEGNAZIONE FRAMES

→ Nella paginazione su richiesta pura, inizialmente tutti I frame sono posti nella lista dei frame liberi, quando il primo processo deve essere caricato in memoria, il processo genera una sequenza di page fault.
→ Un metodo più efficiente, consiste nell'assegnare al processo direttamente in fase di caricamento un certo numero di frame liberi.
→ Ogni processo ha bisogno di un numero minimo di pagine che devono essere assegnate ad ogni processo. Tale numero dipende dal linguaggio macchina.

(m=numero_frame_liberi // n= numero_processi // x=frame_assegnati)

ASSEGNAZIONE UNIFORME

x = m/n

ASSEGNAZIONE PER PRIORITA'

x_1 = (m*priorità_proc)/tot_priorità → se priorità maggiore = numero maggiore

x_2 = [m*(1/priorità_proc)]/(somma_reciproci_ priorità) → priorità maggiroe=numero minore

ASSEGNAZIONE PROPORZIONALE

x = (m*n°_pagine_processo)/n°_pag_tot

GESTIONE DELLA MEMORIA IN LINUX

MEMORIA FISICA

→ più liste di frames liberi a seconda delle dimensione delle “regioni” (regioni da 1, 2,4,8,16,32, …, 2^k pagine)

→quando arriva richiesta di allocazione ‘n’ frames si assegna la prima regione (FIRST FIT) capace di contenerli ($m=2^k$ con $n<k$)

→ quando un processo lascia la RAM prima di inserire la regione libera in una delle liste si vede se ci sono regioni libere adiacenti (buddy). Se ci sono si fondono in una regione più grande

MEMORIA VIRTUALE

→ l’avvicendamento RAM-HD non avviene solo quando non c’è più spazio me ogni volta che il numero di frames liberi scende sotto una soglia fissata

- si selezione il task che occupa più frames
- si sceglie la regione vittima con una lista circolare (così ogni volta gli si toglie una pag a una regione diversa)
- si usano 2 liste (pag attive e inattive) e le pagine possono passare da una all’altra. La vittima è la prima dalla coda delle inattive che ha bit di riferimento a 0