---------- seti 04/11 ----------

scheduling -> nel contesto dei sistemi operativi riguarda il come la CPU / le CPU vengano allocate ai vari processi
esempio: abbiamo un sistema con una sola CPU e un solo core, e abbiamo un'unità di esecuzione e 10 processi che vogliono andare in esecuzione -> lo scheduling fa in modo che il nucleo decida quali processi mandare in esecuzione e per quanto

lo scheduling in generale riguarda il come allocare dei lavori su delle unità di esecuzione
per quanto riguarda il sistema operativo, ci sono due parti che riguardano la schedulazione dei processi:

- il meccanismo che permette al kernel di cambiare processo -> chiamato **cambio di contesto (context switch)** -> scritto in linguaggio macchina
- lo scheduler è la parte del kernel che decide il prossimo processo da mandare in esecuzione, seguendo una certa **politica (policy)** -> scritto in C

nell'esempio con un solo processore da singolo core, se sta girando il processo non sta girando il sistema operativo

come fa il sistema operativo a riprendere il controllo ed eventualmente uccidere il processo che si stava eseguendo e farne partire un altro?
ci sono diversi approcci:

- **cooperativo** -> il sistema operativo si "fida" dei processi
    - i processi faranno dei system call, facendo delle system call chiamano il kernel e a quel punto il kernel può decidere cosa fare con il processo
    - se un processo non fa system call il computer muore lol, rimane per sempre nel processo
- **non cooperativo** -> il sistema operativo riprende il controllo a forza, tramite un timer interrupt

nel **context-switch** si salvano i registri del processo che si stava eseguendo e si vanno a caricare i registri del processo che vogliamo mandare in esecuzione

Ogni processo ha uno stack utente e uno stack kernel

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) from k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Lo stack pointer è un registro che punta alla cima dello stack, però un processo può usarlo come vuole -> lo stack pointer può avere un valore sbagliato, oppure il processo potrebbe decidere di utilizzare lo stack pointer come registro e usarlo per conti, ecc.

Quando arriva un interrupt automaticamente la CPU passa allo stack kernel, che è un valore che ha settato il kernel nell'inizializzazione del sistema -> in questo modo sappiamo quindi che l'interrupt handler (il pezzo di codice che gestisce l'interrupt) avrà uno stack valido

Quindi se vogliamo passare da un processo A ad un processo B:

- Si stava eseguendo il processo A
- Arriva l'interrupt
- Verranno salvati dei registri sul kernel stack del processo A
- Si va ad eseguire l'interrupt handler
- Il kernel decide di passare al processo B
- Si salverà nel PCD (struttura che contiene le informazioni dei processi, in questo momento quelli del processo A) i valori attuali del registro della CPU e andrà a caricare quelli del processo B
- Ritornerà dall'interrupt

Un processo nella sua vita può essere in tanti stati:

- Lo stato ready -> il processo è pronto ad essere eseguito ma non si sta eseguendo
- Lo stato running -> il processo si sta eseguendo
- Lo stato blocked / waiting -> il processo non ha ricevuto un input e sta aspettando (quando ricevuto entrerà in ready)

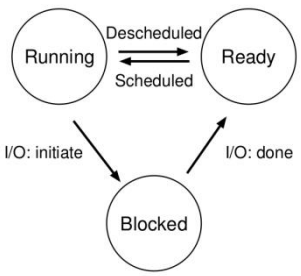


Figure 4.2: **Process: State Transitions**

Le politiche di scheduling sono quegli algoritmi che decidono quale processo mandare in esecuzione -> ci sono delle metriche

Partiamo con assunzioni irrealistiche:

- Ciascun job duri lo stesso tempo
- Tutti i job arrivino allo stesso momento
- Una volta iniziato un job si esegue fino in fondo, senza interruzioni
- Non c'è input/output
- Il tempo di ciascun job è noto a priori

Prima metrica -> **Tempo di turn around** =

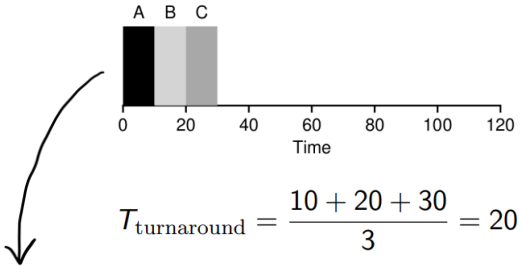
$$tempo\ di\ completamento - tempo\ di\ arrivo$$

Turn-around time

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$$

con le assunzioni iniziali, $T_{\text{arrival}} = 0$ quindi:

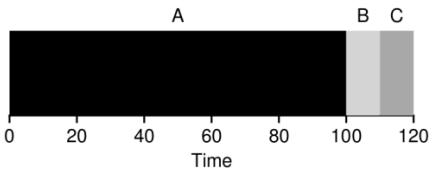
$$T_{\text{turnaround}} = T_{\text{completion}}$$



$$T_{\text{turnaround}} = \frac{10 + 20 + 30}{3} = 20$$

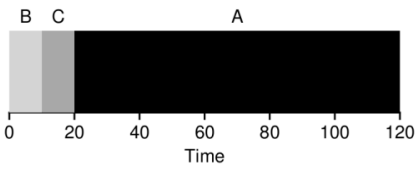
Se A, B e C arrivano allo stesso tempo, l'algoritmo deve scegliere un ordine

Se A dura 100, B e C 10



$$T_{\text{turnaround}} = \frac{100 + 110 + 120}{3} = 110$$

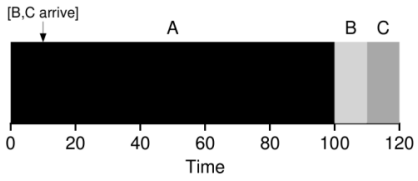
Effetto convoglio: gli altri due job sono corti e il primo è lungo



$$T_{\text{turnaround}} = \frac{10 + 20 + 120}{3} = 50$$

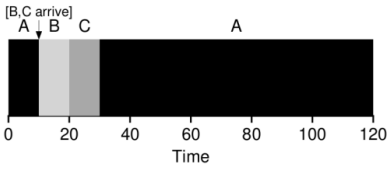
Short job first: algoritmo più ottimale

se non arrivano tutti allo stesso momento
se arriva prima a lungo, e poco dopo arriva b più corto, ormai si sta eseguendo a quindi non possiamo mandare in esecuzione gli altri



$$T_{\text{turnaround}} = \frac{100 + (110 - 10) + (120 - 10)}{3} \approx 103$$

quindi cosa si potrebbe fare? quando arrivano b e c metto in pausa l'esecuzione di a, completo b e c, e poi riprendo a



$$T_{\text{turnaround}} = \frac{120 + (20 - 10) + (30 - 10)}{3} = 50$$

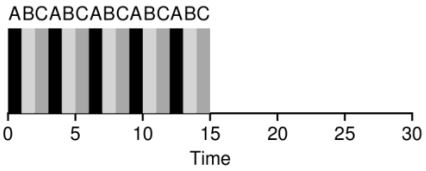
shortest time-to-completion first

può però soffrire di starvation perché se continuano ad arrivare job corti, il job a lungo non finirà mai
ma non può essere davvero usato, in quanto in un sistema operativo non sappiamo quanto sarà lungo un processo

inoltre, mentre nei sistemi non interattivi il tempo di turnaround medio può essere una metrica sensata, nei sistemi interattivi c'è un'altra metrica molto importante, ossia il tempo di risposta, che deve essere molto basso

$$tempo\ di\ risposta = tempo\ di\ inizio\ esecuzione - tempo\ di\ arrivo$$

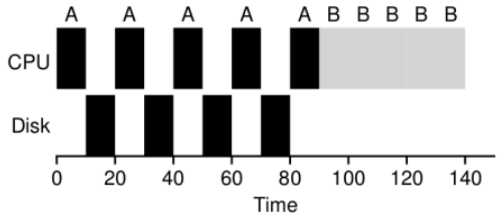
un modo per avere un tempo di risposta buono è il **round-robin** (girotondo) -> mandiamo in esecuzione tutti i job che abbiamo per una piccola fettina di tempo, alternandoli



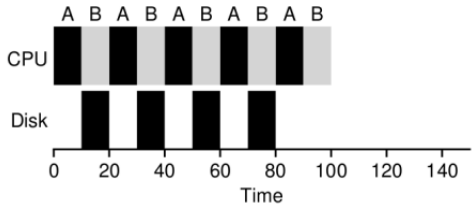
il response-time con il round-robin si può ottimizzare restringendo la fettina di tempo -> non troppo piccola perché se no la cpu fa solo context-switch e non fa altro di utile

- evita la starvation ma penalizza i job corti

Sarebbe stupido tenere allocata la CPU per un processo che attende input/output, quindi lo buttiamo fuori e possiamo usare la CPU per altro



quindi



Esempi di algoritmi "real world"

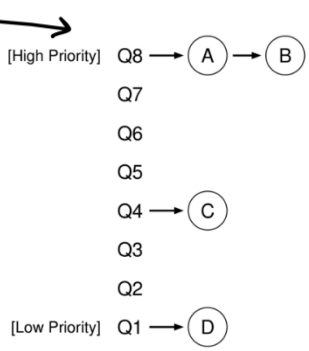
- Multi-level feedback queue (usato da Windows e tempo fa da Linux)
- Proportional share e Linux CFS (Completely Fair Scheduler)

Multi-level feedback queue (MLFQ):

- Cerca di utilizzare sia il turnaround time che il response time -> cerca di eseguire prima i job corti, ma anche ottimizzando response time
- L'idea è quindi che i processi interattivi avranno una priorità tendenzialmente più alta, mentre invece i processi CPU bound (che usano più la CPU) avranno una priorità più bassa
- L'idea del MLFQ è di avere più code a priorità diversa; e per tutti i processi che stanno sulla stessa coda, vanno in round-robin
- La proprietà di un processo verrà variata in base a come si comporta il processo; quello che fa l'algoritmo è cercare di prevedere come si comporterà un processo basandosi su quello che ha fatto fin'ora

- Nell'esempio in foto, abbiamo 8 code, da Q1 a Q8, con Q1 a bassa priorità e Q8 ad alta priorità; le regole dicono
  1. Se $p(A) > p(B)$, gira A (e non gira B)
  2. Se $p(A) = p(B)$, A e B in RR

  con P = priorità
  - Con queste due regole, starvation per C e B



Quindi modifichiamo le regole fino a quelle finali

1. Se $p(A) > p(B)$, gira A (e non gira B)
2. Se $p(A) = p(B)$, A e B in RR
3. Un nuovo job entra con la priorità massima
4. Quando un job usa un tempo fissato t a una certa priorità x (considerando la somma dei tempi usati nella coda x), allora la sua priorità viene ridotta
5. Ogni s secondi, spostiamo tutti i job alla priorità più alta

Il problema più grosso dell'algoritmo è definire i parametri (T, X, ecc.)

Completely Fair Scheduler (CFS):

- Se io ho 3 processi che vogliono usare la CPU, do $1/3$ di CPU a testa
- In generale manderò in esecuzione il processo che ha usato la CPU per meno tempo -> perché se ne ha usata meno "se ne merita ancora" per essere pari agli altri
- Il CFS misura il tempo utilizzato sulla CPU tramite **virtual runtime** -> **vruntime**
- Se tutti i processi avessero la stessa identica priorità, il virtual runtime sarebbe semplicemente il runtime (quanto tempo un processo ha speso usando la CPU); il virtual run time ci permette di avere processi con più priorità, in cui facciamo degli sconti, in cui lasciamo usare più CPU perché ha più priorità -> gli viene conteggiato meno il tempo in cui sta sulla CPU
- In generale, quando c'è da scegliere quale processo mandare in esecuzione, il CFS sceglie quello che ha il virtual runtime più piccolo
- CFS usa vari parametri, uno è *sched_latency* -> il tempo assegnato ad ogni processo
- Se ci sono n processi che sono pronti ad essere eseguiti, quindi nello stato ready, la fettina di tempo che viene data a ogni processo è $\frac{sched\_latency}{n}$ -> se però è troppo piccolo, vado a prendere il valore di un altro parametro, $min\_granularity$
- Vengono mandati in esecuzione questi processi, e poi il vruntime di ogni processo verrà aggiornato in base a quanto tempo sono stati effettivamente in esecuzione

$$\texttt{slice}_k = \max(\frac{\texttt{sched\_latency}}{n}, \texttt{min\_granularity})$$

$$\texttt{vruntime}_k \mathrel{+}= \texttt{runtime}_k$$

- Nei sistemi reali, alcuni processi sono più importanti di altri, quindi c'è un valore ***nice*** che indica la priorità di un processo -> più un processo è ***nice***, più cede la CPU ad altri

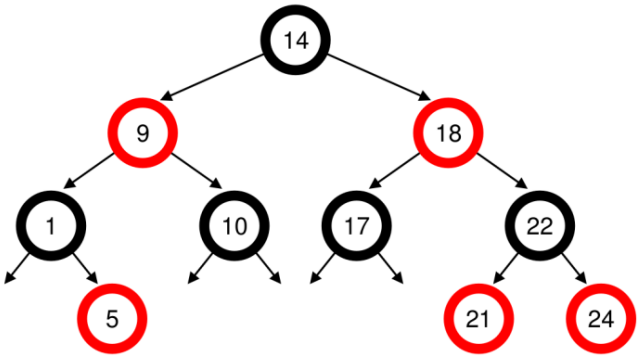
$$\texttt{slice}_k = \max(\texttt{sched\_latency} \cdot \frac{w_k}{\sum_{i=0}^{n-1} w_i}, \texttt{min\_granularity})$$

$$\texttt{vruntime}_k \mathrel{+}= \frac{\texttt{runtime}_k}{w_k}$$

dove

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

- per trovare efficientemente il minimo/aggiornare il vruntime dei processi pronti, essi vengono tenuti in un albero binario bilanciato di tipo rosso/nero:



con costo log(n)

- che vruntime devo dare ad un processo nuovo?
  il vruntime minimo che c'è nel sistema -> quindi non è "completely fair" cin i processi che fanno frequentemente I/O