Scheduling della CPU

- Concetti base
- Criteri di Scheduling
- Algoritmi di Scheduling

Scheduling di processi

- Obbiettivo della multiprogrammazione:
  esecuzione contemporanea di alcuni processi in modo da massimizzare l'uso della CPU
- Obbiettivo del time-sharing:
  commutare l'uso della CPU tra diversi processi in modo da far interagire gli utenti con ciascun programma in esecuzione
- Lo **scheduler** gestisce l'avvicendamento dei processi in CPU
  - Decide quale processo deve essere in esecuzione ogni istante
  - interviene quando viene richiesta un'operazione di I/O e quando un'operazione di I/O termina, ma anche periodicamente per assicurare il buon funzionamento del sistema

Tipi di Scheduler

- Lo *scheduler di lungo termine* seleziona quali processi creare (e quindi aggiungere alla coda ready) fra quelli che non hanno ancora iniziato la loro esecuzione.
  - Viene usato nei sistemi batch.
  - Nei sistemi interattivi tipo Unix, appena lanciato il programma viene creato il relativo processo
- Lo *scheduler di breve termine* seleziona quali processi ready devono essere eseguiti, e quindi assegna la CPU.
- Lo *scheduler di medio termine* gestisce i procesi bloccati per lunghe attese. L'immagine di tali processi può venire copiata in memoria secondaria (disco) al fine di ottimizzare l'uso della memoria centrale (swap-out)

Dispatcher

- Il *dispatcher* è il modulo che dà il controllo della CPU al processo selezionato dallo scheduler di breve termine. Questo comporta
  - switch di contesto
  - passaggio della CPU da modo supervisore a modo user
  - salto alla locazione del programma utente per riprendere il processo
- È essenziale che sia veloce
- La *latenza di dispatch* è il tempo necessario per fermare un processo e riprenderne un altro

Quando interviene lo scheduler a breve termine?

- Eventi che possono causare l'intervento dello scheduler (e un possibile context switch)

  1. quando un processo passa da running a waiting
     (system call bloccante, operazione I/O)
  2. quando un processo passa da running a ready
     (a causa di un interrupt)
  3. quando un processo passa da waiting a ready
  4. termina

- nelle condizioni 1 e 4 l'unica scelta è selezionare un'altro processo (occorre quindi effettuare un context switch)
- nelle condizioni 2 e 3 è possibile continuare ad eseguire lo stesso processo

Tipi di scheduler

- Scheduler **non-preemptive** o **cooperativo**
  - Se i context switch avvengono solo nelle condizioni 1 e 4
  - Cioè: il controllo della risorsa viene trasferito solo se l'assegnatario attuale lo cede volontariamente
  - Es. Windows 3.1, Mac OS versione < 8
- Scheduler **preemptive**
  - Se i context switch possono avvenire in ogni condizione
  - Cioè: è possibile che il controllo della risorsa venga tolto all'assegnatario attuale a causa di un evento
  - Es. tutti gli scheduler moderni

Tipi di scheduler

- Vantaggi dello scheduling **cooperativo**
  - non richiede meccanismi hardware come ad esempio timer programmabili
- Vantaggi dello scheduling **preemptive**
  - permette di utilizzare al meglio le risorse

Criteri di scelta di uno scheduler

- *Utilizzo della CPU:*
  - percentuale di tempo in cui la CPU viene utilizzata per eseguire processi
  - deve essere massimizzato
- *Throughput (produttività):*
  - numero di processi completati nell'unità di tempo
  - dipende dalla lunghezza dei processi
  - deve essere massimizzato

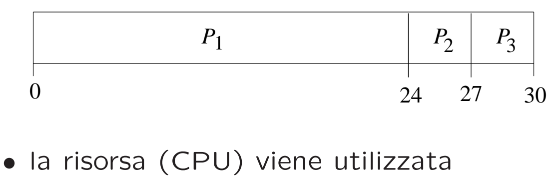
- *Tempo di turnaround (completamento):*
  - tempo che intercorre dalla creazione di un processo alla sua terminazione (include tempi di attesa)
  - deve essere minimizzato
- *Tempo di attesa:*
  - il tempo trascorso da un processo nella coda ready
  - deve essere minimizzato
- *Tempo di risposta:*
  - tempo he intercorre fra la creazione e il tempo della prima risposta (è pensato per sistemi time-sharing)
  - deve essere minimizzato

Caratteristiche dei processi

- Durante l'esecuzione di un processo
  - si alternano periodi di attività svolte dalla CPU (*CPU burst*)
  - e periodi di attività di I/O (*I/O burst*)
- I processi
  - caratterizzati da CPU burst molto lunghi si dicono *CPU bound*
  - caratterizzati da I/O burst molto lunghi si dicono *I/O bound*

Diagrammi di Gantt

- Per rappresentare uno schedule si usani i diagrammi di Gantt
- Ad esempio nel diagramma



- la risorsa (CPU) viene utilizzata
  - dal processo P_1 dal tempo 0 al tempo 24
  - dal processo P_2 dal tempo 24 al tempo 27
  - dal processo P_3 dal tempo 27 al tempo 30

Algoritmi di Scheduling

- First-Come First-Served (FCFS)
- Shortest Job First
- Scheduling con priorità
- Round Robin
- Scheduling con code multiple (e feedback)
- Schedulazione garantita
- Schedulazione a lotteria
- Scheduling multi-processore
- Scheduling real-time
- Esempi: Scheduling Unix e Windows 2000

Scheduling First-Come, First-Served (FCFS)

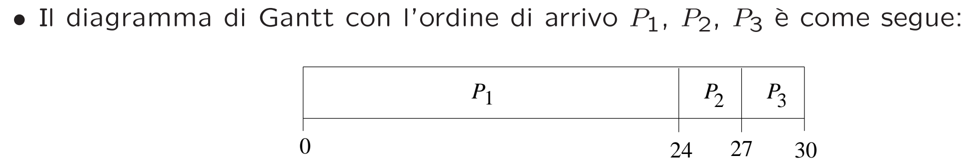
- Algoritmo
  - Il processo che arriva per primo viene servito per primo
  - politica senza preemption
- Implementazione
  - tramite una coda (politica FIFO)
- Problemi
  - elevati tempi medi di attesa e turnaround (tempo che intercorre tra creazione a terminazione)
  - i processi CPU bound possono ritardare i processi I/O bound

FCFS - Esempio

- Consideriamo i seguenti processi e tempi di esecuzione in CPU:

Processo	Burst Time
P_1	24
P_2	3
P_3	3

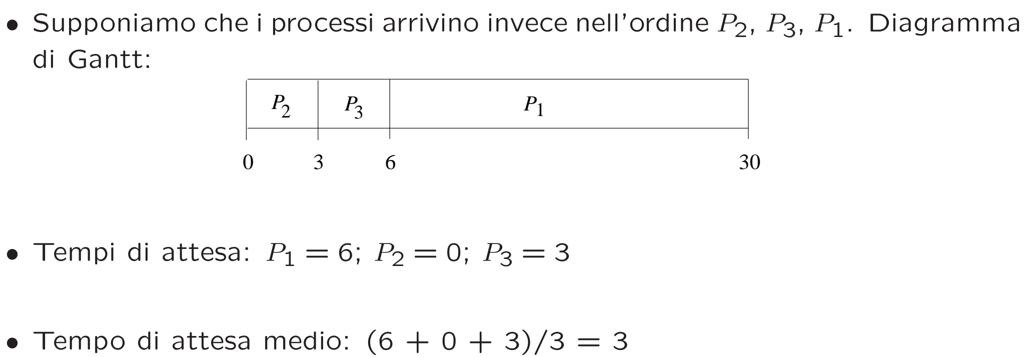
- Il diagramma di Gantt con l'ordine di arrivo P_1, P_2, P_3 è come segue:



- Tempi di attesa: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Tempo di attesa medio: $(0 + 24 + 27)/3 = 17$

FCFS - Esempio

- Supponiamo che i processi arrivino invece nell'ordine P_2, P_3, P_1. Diagramma di Gantt:



- Tempi di attesa: $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Tempo di attesa medio: $(6 + 0 + 3)/3 = 3$

FCFS - Esempio

- Supponiamo di avere un processo CPU bound P_1
- un certo numero di processi I/O bound P_2 P_3 ...
- i processi I/O bound si mettono in coda dietro al processo CPU bound, ed in alcuni casi la ready queue si può svuotare

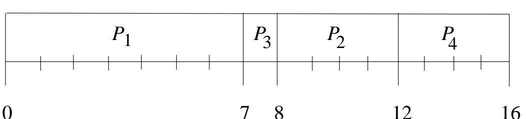
- *Effetto convoglio*:
  - $P1$, $P2$, $P3$, ... entrano in coda in sequenza
  - $P1$ esegue in CPU per un lungo periodo (CPU bound!)
  - $P1$ esegue una operazione di I/O (lascia la CPU)
  - $P2$ va in CPU, esegue subito un'operazione di I/O (I/O bound!) e quindi lascia la CPU
  - $P3$ va in CPU, esegue subito un'operazione di I/O, lascia la CPU ...
  - la coda ready è vuota e quindi la CPU rimane inutilizzata
  - $P1$ ha terminato l'operazione ed esegue di nuovo in CPU
  - ...

Scheduling Shortest-Job-First (SJF)

- Algoritmo
  - Si associa ad ogni processo la lunghezza del suo prossimo burst di CPU. I processi vengono ordinati e schedulati per tempi crescenti.
  - SJF è ottimale: fornisce il minimo tempo di attesa per un dato insieme di processi.
- È impossibile da implementare in pratica (non si può predire il tempo di CPU necessario ad un programma!)
- Si possono fornire solo delle approssimazioni
- Si può verificare starvation

Esempio di SJF

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Tempo di attesa medio = (0 + 6 + 3 + 7)/4 = 4

Come determinare la lunghezza del prossimo ciclo di burst?

- Si può solo dare una *stima*
- Nei sistemi batch, il tempo viene stimato dagli utenti
- Nei sistemi time sharing, possono essere usati i valori dei burst precedenti, con una media pesata esponenziale
  1. t_n = tempo dell'n-esimo burst di CPU
  2. τ_{n+1} = valore previsto per il prossimo burst di CPU
  3. α parametro, $0 \leq \alpha \leq 1$
  4. Calcolo:

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

Esempi di media esponenziale

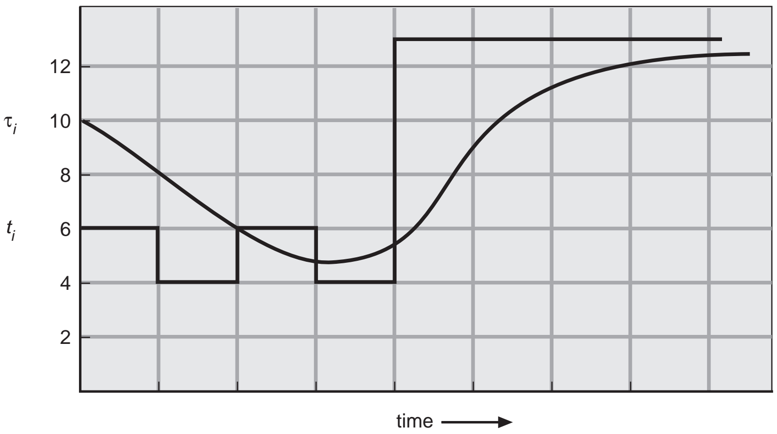
- Espandendo la formula:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\tau_0$$

- Se $\alpha = 0$: $\tau_{n+1} = \tau_0$
  - la storia recente non conta
- Se $\alpha = 1$: $\tau_{n+1} = t_n$
  - Solo l'ultimo burst conta
- Valore tipico per α: 0.5; in tal caso la formula diventa

$$\tau_{n+1} = \frac{t_n + \tau_n}{2}$$

Predizione con media esponenziale



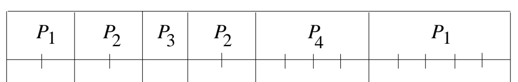
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Approssimazioni di SJF

- Esistono due versioni per le approssimazioni di SJF (basate ad esempio sulla predizione del CPU burst)
  - Non preemptive:
    Il processo corrente esegue fino al completamento del suo CPU burst
  - Preemptive o Shortest-Remaining-Time First (SRTF)
    Il processo corrente può essere messo nella coda ready, se arriva un processo con un CPU burst più breve di quanto rimane da eseguite al processo corrente

Esempio di SRTF

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Tempo di attesa medio = (9 + 1 + 0 + 2)/4 = 3

Scheduling a priorità

- Un numero (intero) di priorità è associato ad ogni processo
- La CPU viene allocata al processo con la priorità più alta (intero più piccolo $\equiv$ priorità più grande)
- Le priorità possono essere definite
  - internamente: in base a parametri misurati dal sistema sul processo (tempo di CPU impiegato, file aperti, memoria, interattività, uso di I/O...)
  - esternamente: importanza del processo, dell'utente proprietario, dei soldi pagati, ...
- Gli scheduling con priorità possono essere preemptive o nonpreemptive
- SJF è uno scheduling a priorità, dove la priorità è il prossimo burst di CPU previsto

Scheduling con priorità (cont.)

- Problema: *starvation* – i processi a bassa priorità possono venire bloccati da un flusso continuo di processi a priorità maggiore
  - vengono eseguiti quando la macchina è molto scarica
  - oppure possono non venire mai eseguiti
- Soluzione: invecchiamento (*aging*) – con il passare del tempo, i processi non eseguiti aumentano la loro priorità

Round Robin (RR)

- Algoritmo
  - È basato sul concetto di quanto di tempo (time-slice)
  - Un processo non può rimanere in esecuzione per un tempo superiore alla durata del quanto di tempo (tipicamente 10-100 millisecondi)

RR - Implementazione

- L'insieme dei processi ready è organizzato come una coda
- Vi sono due possibilità:
  - un processo rilascia volontariamente la CPU (ad esempio operazione I/O)
  - un processo esaurisce il suo quanto di tempo sena completare il suo CPU burst, nel qual caso viene aggiunto in fondo alla coda dei processi ready
- In entrambi i casi il prossimo processo da eseguire è il primo della coda ready
- Se ci sono n processi in ready, e il quanto è q, allora ogni processo riceve $1/n$ del tempo di CPU in periodi di durata massima q. Nessun processo attende più di $(n-1)q$

Esempio: RR con quanto = 20

Processo	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- Diagramma di Gantt

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

- Tipicamente, si ha un tempo di turnaround medio maggiore, ma minore tempo di risposta

Prestazioni dello scheduling Round-Robin

- q grande $\Rightarrow$ degenera nell'FCFS
- q piccolo $\Rightarrow$ q deve comunque essere grande rispetto al tempo di context switch, altrimenti l'overhead è elevato

process time = 10	quantum	context switches
0 – 10	12	0
0 – 6 – 10	6	1
0 1 2 3 4 5 6 7 8 9 10	1	9

- L'80% dei CPU burst dovrebbero essere inferiori a q

Prestazioni dello scheduling Round-Robin (Cont.)

process	time
P_1	6
P_2	3
P_3	1
P_4	7

Scheduling con code multiple

- La coda di ready è partizionata in più code separate: ad esempio, processi "foreground" (interattivi), processi "background" (batch)

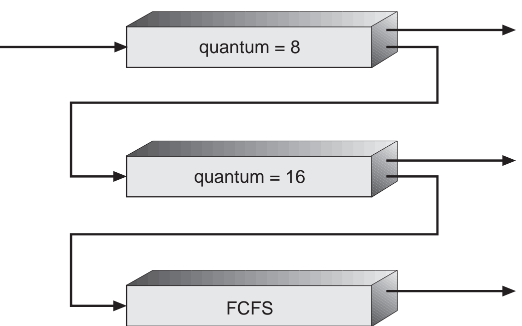
Scheduling con code multiple (Cont.)

- Ogni coda ha un suo algoritmo di scheduling; ad esempio, RR per i foreground, FCFS o SJF per i background
- Lo scheduling deve avvenire tra tutte le code: alternative
  - Scheduling a priorità fissa: eseguire i processi di una coda solo se le code di priorità superiore sono vuote.
    $\Rightarrow$ possibilità di starvation.
  - Quanti di tempo per code: ogni coda riceve un certo ammontare di tempo di CPU per i suoi processi; ad es., 80% ai foreground in RR, 20% ai background in FCFS

Scheduling a code multiple con feedback

- I processi vengono spostati da una coda all'altra, dinamicamente.
- Aging: se un processo ha usato recentemente
  - molta CPU, viene spostato in una coda a minore priorità
  - poca CPU, viene spostato in una coda a maggiore priorità
- Uno scheduler a code multiple con feedback viene definito dai seguenti parametri:
  - numero di code
  - algoritmo di scheduling per ogni coda
  - come determinare quando promuovere un processo
  - come determinare quando degradare un processo
  - come determinare la coda in cui mettere un processo che entra nello stato di ready

Esempio di code multiple con feedback



Tre code:
- Q_0 – quanto di 8 msec
- Q_1 – quanto di 16 msec
- Q_2 – FCFS

Scheduling:

- Un nuovo job entra in Q_0, dove viene servito FCFS con prelazione. Se non termina nei suoi 8 millisecondi, viene spostato in Q_1.
- Nella coda Q_1, ogni job è servito FCFS con prelazione, quando Q_0 è vuota. Se non termina in 16 millisecondi, viene spostato in Q_2.
- Nella coda Q_2, ogni job è servito FCFS senza prelazione, quando Q_0 e Q_1 sono vuote.

Schedulazione garantita

- Si promette all'utente un certo quality of service (che poi deve essere mantenuto)
- Esempio: se ci sono n utenti, ad ogni utente si promette $1/n$ della CPU.
- Implementazione:
  - per ogni processo T_p si tiene un contatore del tempo di CPU utilizzato da quando è stato lanciato.
  - il tempo di cui avrebbe diritto è $t_p = T/n$, dove $T =$ tempo trascorso dall'inizio del processo.
  - priorità di $P = T_p/t_p$ — più è bassa, maggiore è la priorità

Schedulazione a lotteria

- Semplice implementazione di una schedulazione “garantita”
  - Esistono un certo numero di “biglietti” per ogni risorsa
  - Ogni utente (processo) acquisisce un sottoinsieme di tali biglietti
  - Viene estratto casualmente un biglietto, e la risorsa viene assegnata al vincitore
- Per la legge dei grandi numeri, alla lunga l’accesso alla risorsa è proporzionale al numero di biglietti
- I biglietti possono essere passati da un processo all’altro per cambiare la priorità (esempio: client/server)

Scheduling multi-processore (cenni)

- Lo scheduling diventa più compesso quando più CPU sono disponibili
- Sistemi *omogenei:* è indiff. su quale processore esegue il prossimo task
- Può comunque essere richiesto che un certo task venga eseguito su un preciso processore (*pinning*)
- Bilanciare il carico (*load sharing*) ⇒ tutti i processori selezionano i processi dalla stessa ready queue
- problema di accesso condiviso alle strutture del kernel
  - *Asymmetric multiprocessing (AMP):* solo un processore per volta può accedere alle strutture dati del kernel — semplifica il problema, ma diminuisce le prestazioni (carico non bilanciato)
  - *Symmetric multiprocessing (SMP):* condivisione delle strutture dati. Serve hardware particolare e di controlli di sincronizzazione in kernel

Scheduling Real-Time

- *Hard real-time:* si richiede che un task critico venga completato entro un tempo ben preciso e garantito.
  - prenotazione delle risorse
  - determinazione di tutti i tempi di risposta: non si possono usare memorie virtuali, connessioni di rete, . . .
  - solitamente ristretti ad hardware dedicati
- *Soft real-time:* i processi critici sono prioritari rispetto agli altri
  - possono coesistere con i normali processi time-sharing
  - lo scheduler deve mantenere i processi real-time prioritari
  - la latenza di dispatch deve essere la più bassa possibile
  - adatto per piattaforme general-purpose, per trattamento di audio-video, interfacce real-time, . . .

Minimizzare il tempo di latenza

- Un kernel *non prelazionabile* è inadatto per sistemi real-time: un processo non può essere prelazionato durante una system call
  - *Punti di prelazionabilità (preemption points):* in punti “sicuri” delle system call di durata lunga, si salta allo scheduler per verificare se ci sono processi a priorità maggiore
  - *Kernel prelazionabile:* tutte le strutture dati del kernel vengono protette con metodologie di sincronizzazione (semafori). In tal caso un processo può essere sempre interrotto.
- *Inversione delle priorità:* un processo ad alta priorità deve accedere a risorse attualmente allocate da un processo a priorità inferiore.
  - *protocollo di ereditarietà delle priorità:* il processo meno prioritario eredita la priorità superiore finché non rilascia le risorse.

Scheduling di breve termine in Unix tradizionale

(fino a 4.3BSD e SVR3)

- a code multiple, round-robin
- ogni processo ha una priorità di scheduling; numeri più grandi indicano priorità minore
- Feedback negativo sul tempo di CPU impiegato
- Invecchiamento dei processi per prevenire la starvation
- Quando un processo rilascia la CPU, va in *sleep* in attesa di un *event*
- Quando l'evento occorre, il kernel esegue un *wakeup* con l'indirizzo dell'evento e il processo *asleep* in testa alla coda sull'evento viene messo nella coda di ready (*)
- I processi che erano in attesa di un evento in modo kernel rientrano con priorità *negativa* e non soggetta a invecchiamento

- (*) Nota:
  - In Unix tradizionale gli eventi sono mappati in indirizzi del kernel
  - Eventi diversi possono essere mappati nello stesso indirizzo kernel (ad es. attesa su un buffer e attesa di completamento di I/O vengono mappati sull'indirizzo del buffer)
  - Più processi possono essere asleep sullo stesso indirizzo kernel
  - Il kernel non tiene traccia di quanti processi sono in attesa
  - A seguito di una wakeup *tutti i processi in attesa* su un evento vengono risvegliati e spostati nello stato Ready
  - Tuttavia molti di essi torneranno subito asleep

Scheduling in Unix tradizionale (Cont.)

- 1 quanto = 5 o 6 tick = 100 msec
- alla fine di un quanto, il processo viene prelazionato
- quando il processo j rilascia la CPU
  - viene incrementato il suo contatore CPU_j di uso CPU
  - viene messo in fondo alla stessa coda di priorità
  - riparte lo scheduler su tutte le code
- 1 volta al secondo, vengono ricalcolate tutte le priorità dei processi in user mode (dove $nice_j$ è un parametro fornito dall'utente):
  $CPU_j = CPU_j/2$ (*fading esponenziale*)
  $P_j = CPU_j + nice_j$

  I processi in kernel mode non cambiano priorità.

Scheduling in Unix tradizionale (Cont.)

In questo esempio, 1 secondo = 4 quanti = 20 tick

Tempo	Processo A Pr_A	CPU_A	Processo B Pr_B	CPU_B	Processo C Pr_C	CPU_C
0	0	0	0	0	0	0
	0	5	0	0	0	0
	0	5	0	5	0	0
	0	5	0	5	0	5
	0	10	0	5	0	5
1	5	5	2	2	2	2
	5	5	2	7	2	2
	5	5	2	7	2	7
	5	5	2	12	2	7
	5	5	2	12	2	12
2	2	2	6	6	6	6
	2	7	6	6	6	6
	2	12	6	6	6	6
	2	17	6	6	6	6
	2	22	6	6	6	6
3	11	11	3	3	3	3
	11	11	3	8	3	3
	11	11	3	8	3	8
	11	11	3	13	3	8

...

Scheduling in Unix tradizionale (Cont.)

Considerazioni

- Adatto per time sharing generale
- Privilegiati i processi I/O bound - tra cui i processi interattivi
- Garantisce assenza di starvation per CPU-bound e batch
- Quanto di tempo indipendente dalla priorità dei processi
- Non adatto per real time
- Non modulare, estendibile

Inoltre il kernel 4.3BSD e SVR3 non era prelazionabile e poco adatto ad architetture parallele.

Scheduling in Unix moderno (4.4BSD, SVR4 e successivi)

Applicazione del principio di separazione tra il meccanismo e le politiche

- Meccanismo generale
  - 160 livelli di priorità (numero maggiore $\equiv$ priorità maggiore)
  - ogni livello è gestito separatamente, event. con politiche differenti
- *classi di scheduling:* per ognuna si può definire una politica diversa
  - intervallo delle priorità che definisce la classe
  - algoritmo per il calcolo delle priorità
  - assegnazione dei quanti di tempo ai varî livelli
  - migrazione dei processi da un livello ad un altro
- Limitazione dei tempi di latenza per il supporto real-time
  - inserimento di punti di prelazionabilità del kernel con check del flag `kprunrun`, settato dalle routine di gestione eventi

Scheduling in Unix moderno (4.4BSD, SVR4 e successivi)

Assegnazione di default: 3 classi

Real time: possono prelazionare il kernel. Hanno priorità e quanto di tempo fisso.

Kernel: prioritari su processi time shared. Hanno priorità e quanto di tempo fisso. Ogni coda è gestita FCFS.

Time shared: per i processi "normali". Ogni coda è gestita round-robin, con quanto minore per priorità maggiore. Priorità variabile secondo una tabella fissa: se un processo termina il suo quanto, scende di priorità.

Priority Class	Global Value	Scheduling Sequence
Real-time	159 • • • • 100	first ↓
Kernel	99 • • 60	↓
Time-shared	59 • • • • 0	↓ last

Considerazioni sullo scheduling SVR4

- Flessibile: configurabile per situazioni particolari
- Modulare: si possono aggiungere altre politiche (p.e., batch)
- Le politiche di default sono adatte ad un sistema time-sharing generale
- manca(va) uno scheduling real-time FIFO (aggiunto in Solaris, Linux, ...)

Scheduling di Windows 2000

Un thread esegue lo scheduler quando

- esegue una chiamata bloccante
- comunica con un oggetto (per vedere se si sono liberati thread a priorità maggiore)
- alla scadenza del quanto di thread

Inoltre si esegue lo scheduler in modo asincrono:

- Al completamento di un I/O
- allo scadere di un timer (per chiamate bloccanti con timeout)

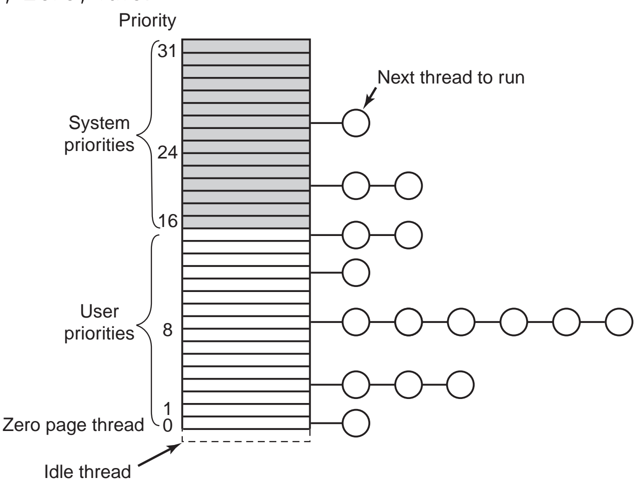
Scheduling di Windows 2000

- I processi possono settare la classe priorità di *processo* (`SetPriorityClass`)
- I singoli thread possono settare la priorità di *thread* (`SetThreadPriority`)
- Queste determinano la *priorità di base* dei thread come segue:

		Win32 process class priorities					
		Realtime	**High**	**Above Normal**	**Normal**	**Below Normal**	**Idle**
Win32 thread priorities	**Time critical**	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Scheduling di Windows 2000

- I thread (NON i processi) vengono raccolti in code ordinate per priorità, ognuna gestita round robin. Quattro classi: *system* ("real time", ma non è vero), *utente*, *zero*, *idle*.



- Solo lo *zero page thread* assume priorità zero (è un thread del kernel che si occupa di ripulire le pagine di memoria quando sono rilasciate da altri thread)
- Lo scheduler sceglie sempre dalla coda a priorità maggiore
- La priorità di un thread utente può essere temporaneamente maggiore di quella base (*spinte*)
  - per thread che attendevano dati di I/O (spinte fino a +8)
  - per dare maggiore reattività a processi interattivi (+2)
  - per risolvere inversioni di priorità