Abbiamo visto gli algoritmi di scheduling. Oggi ne vediamo un altro abbastanza diverso, il Completely Fair Scheduler che è quello usato da Linux (quindi è un esempio del mondo reale).

Gli scheduler proportional-share invece di cercare di ottimizzare i tempi di workaround o response si basano su un concetto semplice: ogni processo dovrebbe avere la sua percentuale di tempo di CPU.

Quello che accade col CFS è che esso conteggia il tempo usando il virtual runtime: vruntime (nel caso priù semplice, è proporzionale al tempo vero di utilizzo della CPU). Quando c’è da scegliere un processo da mandare in esecuzione, CFS seleziona quello col vruntime più piccolo.   
CFS usa vari parametri, uno è il sched\_latency; se ci sono n processi pronti si manda in esecuzione quello col vruntime più basso per un tempo sched\_latency/n, purchè non sia inferiore alla granularità minima (min\_granularity).

Oltretutto, si applica ai processi un fattore di scala, per tenere conto del valore NICE (priorità) di ogni processo, per cui:

sliceK = max(sched\_latency \* wk / Sum(I = 0, n-1) wi, min\_granularity)

vruntimeK += runtime / wK. (si conta il runtime effettivo, perché il processo potrebbe aver fatto I/O durante il suo turno della CPU, e quindi potrebbe non averlo usato tutto).

Per trovare efficientemente il minimo e aggiornare il vruntime dei processi pronti, essi vengono tenuti in un albero rosso/nero (un albero binario che si tiene bilanciato).

Bisogna chiedersi, quando arriva un processo nuovo, cosa gli si dà di vruntime? Non gli si può dare 0, perché sennò il nuovo processo si prende la CPU per troppo tempo (chissà quanto è grande il vruntime del secondo processo più giovane), quindi, quando un processo diventa ready (dopo sleep o init) si prende un vruntime uguale al minimo degli altri: dunque, il processo non è completamente “fair” con i processi che fanno frequentemente I/O (perché un processo che torna dal waiting si ritrova cancellato il tempo che gli era avanzato).

Passiamo al multi-threading.

I thread permettono di avere più flussi di esecuzione in un processo:

* Creare un thread significa creare una “CPU virtuale”
* A livello di kernel: TCB vs PCB. Come esiste il Process Control Block, deve esistere il Thread Control Block (in windows un processo è un contenitore che può avere tanti thread, in linux, viceversa, non fa differenza tra thread e processo a livello di schedulabilità. Il context switch tra thread è più semplice che tra processi, quindi, forse la visione di windows è più efficiente)
* La differenza tra thread e processi è che processi separati girano su spazi di indirizzamento separati e non possono interagire facilmente tra di loro, invece thread separati girano sullo stesso spazio di indirizzamento.

Ogni thread ha il proprio stack, ma condivide codice e dati con altri thread dello stesso processo (quindi, ha il proprio ma può, anche se non si dovrebbe, accedere anche agli altri, siccome sono nello stesso spazio di indirizzamento). Inoltre, ogni thread ha il proprio errno (nel proprio TLS: Thread Local Storage), quindi errno non è esattamente una variabile globale, perché ogni thread ha la sua.

Perché usare i thread? Perché permette di sfruttare il parallelismo (se dobbiamo operare con array molto grossi e ci interessa, chessò, fare la somma di tutti gli elementi, può convenire separare l’array e far lavorare ogni thread su una sotto-sezione di esso per poi mettere insieme gli outputs). Oppure può servire per rispondere a più client contemporaneamente può convenire usare i thread.

Come possiamo creare un thread con l’API Posix? Si usa <pthread.h> (quando compiliamo bisogna aggiungere -pthread sulla linea di comando sennò il linker si incazza). La syscall è in questo caso:

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void\*), void \*arg);

In caso di successo restituisce 0, ma in caso di errore restituisce il codice di errore (contrariamente allo standard POSIX, che prevede di restituire -1 e impostare errno): errno non è impostato.

In pthread\_t \*thread ci troviamo l’identificatore del processo che è appena stao creato.

Il secondo parametro serve a poco, possiamo passarci NULL.

Il terzo parametro è un puntatore a funzione che prende un void\* e restituisce un void\*.

Infine abbiamo un argomento che verrà passato alla start\_routine del terzo argomento. Questo ha senso perché la start\_routine potremmo usarla così per tanti thread diversi. La start\_routine è quindi il main del thread.

**Su ubuntu per avere il man sul thread dobbiamo installare manpage-posix-dev.**

Si può uscire dal thread con un certo valore facendo un return dalla funzione iniziale oppure chiamando pthread\_exit(void \*retval).

Si può attendere la terminazione di un thread con pthread\_join(pthread\_t thread, void \*\*retval);

Può capitare che tra l’accesso in lettura e quello in scrittura di una variabile, lo scheduler passi la cpu a un altro thread, dunque se il context switch avviene mentre un thread ha aperta in lettura una variabile globale che poi verrà modificata da un altro thread, c’è la possibilità che questa variabile venga sovrascritta una volta che il vecchio thread torna in esecuzione.

Una sezione critica è un frammento di codice che accede a una risorsa condivisa. Quando si hanno più flussi di esecuzione, si parla di race conditions, quando il risultato finale dipende dalla temporizzazione o dall’ordine con cui vengono schedulati: in tal caso la computazione è non-deterministica.  
Si ha una race-condition quando più thread eseguono (più o meno allo stesso tempo) una sezione critica.

Per evitare questi problemi, serve sincronizzare fra i thread, attraverso primitive di sincronizzazione (che vedremo in PCAD).

Una soluzione è introdurre dei lock, implementati dai mutex in pthread. Un lock:

si dichiara con pthread\_mutex\_t e si può inizializzare tramite assegnazione della costante PTHREAD\_MUTEX\_INITIALIZER oppure pthread\_muter\_init.  
Può essere acquisito da un thread alla volta tramite pthread\_mutex\_lock.  
Va rilasciato, il prima possibile, tramite pthread\_mutex\_unlock.

Come suggerisce il nome MUTEX, che indica mutua esclusione, i lock sono strutture dati che forniscono accesso esclusivo a qualcosa.

La presa e il rilascio dei lock, tuttavia, causa un ulteriore overhead, quindi l’utilizzo del multithreading è ottimale quando ogni thread si fa i fatti suoi e solo ogni tanto accede a una variabile condivisa. Nell’esempio delle slide usare più thread per incrementare un contatore rende il programma molto più lento.

Se un programma usa più strutture dati (condivise) diverse, probabilmente conviene avere più lock: questo aumenta l’efficienza, ma complica la gestione, quindi non bisogna esagerare (sarebbe eccessivo avere un lock per ogni nodo della lista: ci sarebbe troppo overhead).

Esempi di implementazioni sulle slides.