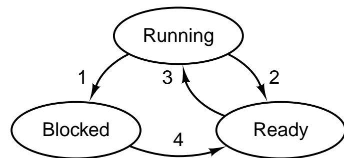
Scheduling

Ho più di un processo in stato *ready*

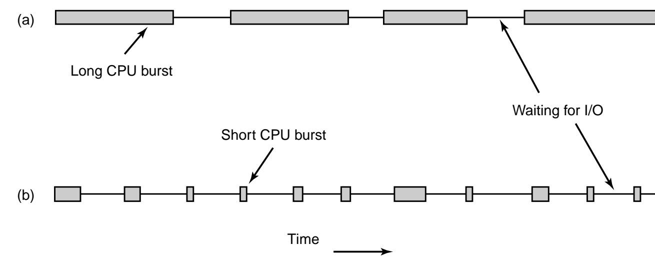
Quale scelgo?

Problema dello *scheduling*



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Periodi di uso della CPU alternati ad attese per I/O



(a) CPU bound

(b) I/O bound

Obiettivi

Tutti i sistemi
- fairness (equità)
- policy enforcement
- max. CPU and I/O utilization

Sistemi a lotti
- max. throughput
- min. turnaround
- max. CPU utilization

Sistemi interattivi
- min. tempi di risposta

Sistemi real-time
- rispettare le deadline

Quando viene eseguito lo scheduling?

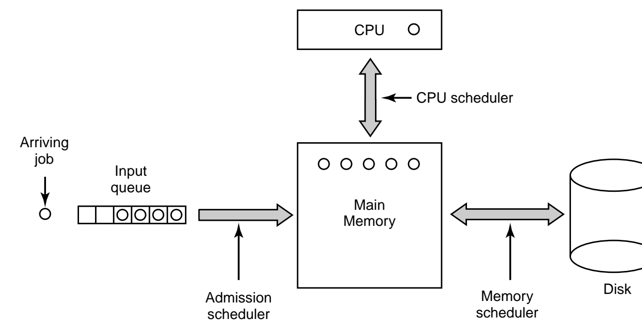
0. creazione/distruzione di un nuovo processo
1. un processo si blocca per fare I/O
2. arriva un interrupt per I/O
3. clock interrupt
4. (Linux) all'uscita da una system call

Senza prelazione (non-preemptive):
un processo esegue fino a che non cede volontariamente la CPU

Con prelazione (preemptive):
un processo in esecuzione può essere bloccato dal S.O.

Scheduling a tre livelli

0. admission scheduler (batch only)
1. memory scheduler
2. CPU scheduler



Scheduling in batch systems I

First come, first served (senza prelazione)

Il processo esegue fino a che non si blocca; poi torna in fondo alla coda

- facile da implementare
- problema: un processo CPU bound rallenta 100 processi I/O bound

Scheduling in batch systems II

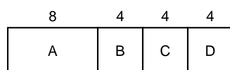
Shortest job first (senza prelazione)

Presumiamo di conoscere in anticipo la durata dei processi

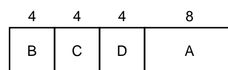
Come *first come, first served* ma la coda è ordinata per durata crescente

Shortest job first

(i processi vengono eseguiti da destra a sinistra)



(a)



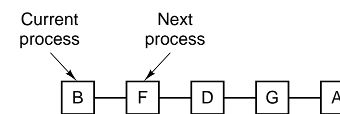
(b)

average turnaround time

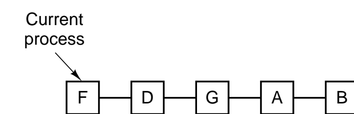
$$(a) \quad \frac{4+8+12+20}{4} = 11$$
$$(b) \quad \frac{8+12+16+20}{4} = 14$$

Scheduling in interactive systems

Round-robin



(a)



(b)

lista di processi *ready*

Quando il processo esaurisce il *quanto* torna in fondo alla lista

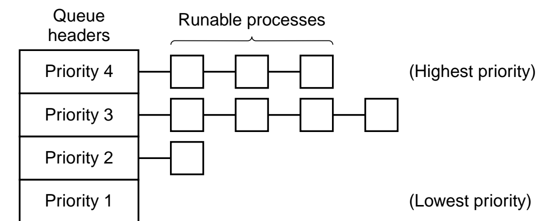
Priority scheduling

Assegnamo una priorità ai processi

Scegliamo sempre il processo di priorità massima

Problema: starvation
⇒ priorità dinamiche
- privilegiare i processi I/O bound

Priority classes



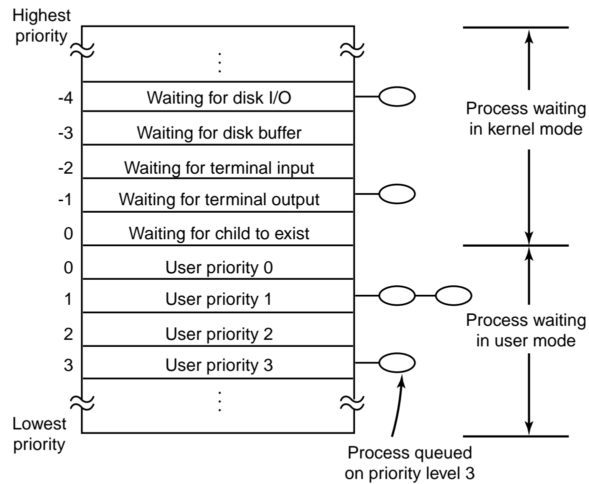
Policy versus Mechanism

Il kernel implementa un meccanismo di scheduling parametrizzabile

I processi possono modificare i parametri (policy)

Scopo: ottimizzare la performance conoscendo il comportamento dei processi

Scheduling in UNIX



Scheduling in UNIX

Code multiple ordinate per *priorità*

algoritmo round-robin all'interno della coda

quanto di tempo

UNIX: Calcolo della priorità

ricalcolata ogni secondo

priority = CPU_usage + nice + base

CPU_usage
- incrementato ad ogni clock tick
- diminuisce col tempo

nice
- range -20 – +20
- default value 0
- solo *root* può assegnare valori negativi

base
- serve a dare priorità ai proc. in kernel mode

Scheduling in Linux

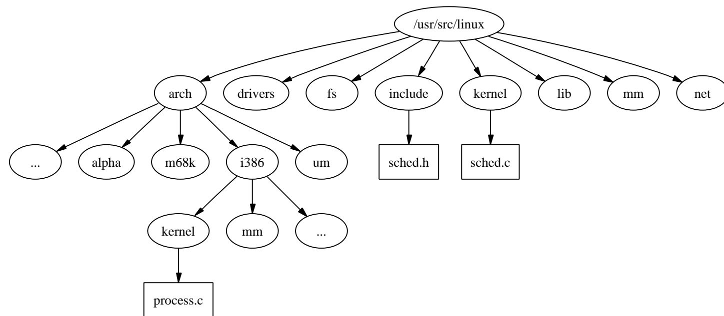
Linux usa i thread a livello di kernel

Tre classi di scheduling

- Real-time FIFO (SCHED_FIFO)
- Real-time round-robin (SCHED_RR)
- Timesharing (SCHED_OTHER)

(si tratta di *soft* real-time!)

Linux source code organization



File rilevanti per lo scheduling

- include/sched.h
- kernel/sched.c
- arch/i386/kernel/process.c

I thread in Linux

ogni thread ha

- una *priorità*
- un quanto

$1 \leq priority \leq 40$

più alto il valore di *priority* migliore il servizio

default: 20

La chiamata di sistema nice(2)

```
int nice(int inc);
```

nice adds inc to the nice value for the calling pid. (A large nice value means a low priority.) — manuale di nice(2)

L'effetto di nice(n) è

$priority := 20 - n$

Il quanto

quantum: numero di clock tick per cui il thread può proseguire

clock tick: 10ms

Calcolo della *goodness*

pseudocodice:

```
if (policy is real-time)
   goodness = 1000 + priority

if (policy is timesharing && quantum > 0)
   goodness = quantum + priority

if (policy is timesharing && quantum == 0)
   goodness = 0
```

bonus per processi che eseguono sulla stessa CPU

bonus per l'ultimo processo che ha eseguito

Calcolo della *goodness*

implementato in kernel/sched.c

```
/*
 * This is the function that decides how desirable a process is..
 * You can weigh different processes against each other depending
 * on what CPU they've run on lately etc to try to handle cache
 * and TLB miss penalties.
 *
 * Return values:
 *       -1000: never select this
 *           0: out of time, recalculate counters (but it might still be
 *              selected)
 *         +ve: "goodness" value (the larger, the better)
 *       +1000: realtime process, select this.
 */
static inline int goodness(struct task_struct * p, int this_cpu, ...)
{
        int weight = -1;
```



```
        /*
         * Non-RT process - normal case first.
         */
        if (p->policy == SCHED_OTHER) {
                /*
                 * Give the process a first-approximation goodness value
                 * according to the number of clock-ticks it has left.
                 *
                 * Don't do any other calculations if the time slice is
                 * over..
                 */
                weight = p->counter;
                if (!weight)
                        goto out;
#ifdef CONFIG_SMP
                /* Give a largish advantage to the same processor...   */
                /* (this is equivalent to penalizing other processors) */
                if (p->processor == this_cpu)
                        weight += PROC_CHANGE_PENALTY;
#endif
                weight += 20 - p->nice;
                goto out;
        }
```

```
        /*
         * Realtime process, select the first one on the
         * runqueue (taking priorities within processes
         * into account).
         */
        weight = 1000 + p->rt_priority;
out:
        return weight;
}
```


Torniamo all'algoritmo di scheduling di Linux...

In ogni istante, si sceglie il thread che ha la miglior *goodness*

Ad ogni tick, si decrementa il quanto

Quando tutti i thread *ready* hanno quantum == 0, ricalcolo *tutti* i quanti

$$quantum = quantum/2 + priority$$

Proprietà dell'algoritmo di scheduling

i processi I/O bound vengono preferiti

i processi CPU bound vengono preferiti in proporzione alla priority

Come cambiare la policy del processo?

```
int sched_setscheduler(pid_t pid, int policy, struct sched_param *p);
```

sched_setscheduler sets both the scheduling policy and the associated parameters for the process identified by pid. [...]

Currently, the following three scheduling policies are supported under Linux: SCHED_FIFO, SCHED_RR, and SCHED_OTHER; their respective semantics is described below...

— manuale di sched_setscheduler(2)

Il nuovo algoritmo di scheduling $O(1)$

Lo scheduler è sostanzialmente lo stesso
da Linux 1.0 a 2.4

Esegue in tempo $O(N)$, $N =$ numero dei processi

Va bene per uso desktop, meno bene per grandi server
$\Rightarrow$ in particolare le app. Java soffrono

Ingo Molnar, autore dello scheduler $O(1)$ in Linux 2.6

Ricompilazione e installazione del kernel

0. scaricare la versione corrente da kernel.org
1. eseguire "make xconfig" o "make menuconfig" per configurare
2. compilare con "make bzImage"
3. copiare il file del kernel da arch/i386/boot/bzImage a /boot/il-mio-kernel
4. editare /etc/lilo.conf
5. eseguire "lilo"
6. eseguire "reboot" per testare il nuovo kernel

vedi il "Kernel-HOWTO" per maggiori dettagli