

Elements de Sistemes Operatius

Capítol 6

La planificació

J. Fornés M. Farreras Y. Becerra

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Grau en Enginyeria Informàtica — Sistemes Operatius

Versió de prova per a lectors · 28 de setembre de 2026



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

© Universitat Politècnica de Catalunya

Aquesta obra està subjecta a la llicència de Creative Commons Reconeixement-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0):

«By Universitat Politècnica de Catalunya — BarcelonaTech (UPC), Any 2026»



<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.ca>

Aquest text s'ha escrit amb l'ajuda de Claude, el model de llenguatge d'Anthropic. Tots els errors que quedin són dels autors.

6 La planificació

Al capítol 5 vam anar deixant coses per a més endavant. Que hi ha una cua de processos preparats i que el `fork()` hi afegeix el fill; que el procés passa de RUN a READY sense demanar-ho ell; que hi ha una interrupció de rellotge que no deixa que ningú monopolitzi la màquina. Tot això és aquest capítol.

Diguem d'entrada què hi busquem. Hi va haver una època en què la planificació era el cor d'un sistema operatiu, perquè la CPU era el recurs escàs i car de la màquina. Avui la CPU sobra sovint i el que escasseja és una altra cosa — la memòria, l'ample de banda, la bateria —, i els planificadors de debò han esdevingut prou complicats perquè estudiar-los línia a línia no compensi. El que sí que compensa, i molt, són les dues o tres idees que hi ha a sota, perquè apareixen a tot arreu: què costa canviar de procés, per què cal un límit de temps, i què vol dir que un repartiment sigui just.

6.1 Política i mecanisme

La distinció que ordena tot el capítol és aquesta, i és una de les grans idees dels sistemes operatius [1].

La **política** decideix *què* s'ha de fer: a qui li toca la CPU ara, i durant quant. El **mecanisme** és *com* es duu a terme aquella decisió: com es desa l'estat d'un procés i es restaura el d'un altre. Són coses separades a posta. El mecanisme és car de fer i no canvia gairebé mai; la política es pot canviar — i de fet Linux n'ha canviat unes quantes vegades — sense tocar ni una línia del mecanisme.

Al planificador — l'*scheduler* — li correspon la política. Al canvi de context, el mecanisme.

6.2 Què es reparteix de debò

Abans de mirar on el kernel es guarda els processos, cal dir bé què és el que reparteix, perquè la resposta que ve al cap és mitja veritat.

La resposta que ve al cap és «processos». Al capítol 12 l'haurem de corregir una mica — el que el planificador tria és un thread, i un procés amb un sol thread és les dues coses alhora —, però la correcció que compta és una altra, i és aquesta.

Mireu la vida d'un procés de prop. No és un bloc de feina que dura des del `fork()` fins a l'`exit()`: és una alternança. Calcula una estona, demana alguna cosa a un dispositiu i s'espera; quan la té, torna a calcular una estona, i torna a demanar. Cada tros de càlcul seguit té nom: és una *ràfega de CPU*, en anglès *CPU burst*.

I una ràfega no s'acaba perquè sí. S'acaba d'una de dues maneres, que són exactament les dues del capítol 9: o bé el procés fa una crida que el bloqueja — un `read()` d'un terminal, un `write()` a una pipe plena — i deixa la CPU ell mateix, o bé no s'acaba i l'hi han de prendre.

El planificador no reparteix processos, ni tan sols threads: reparteix **ràfegues de CPU**. Cada vegada que s'executa, la pregunta que contesta és quina ràfega va ara; de quin procés és, és una conseqüència, no la pregunta.

Això no és una precisió d'erudit: canvia el significat de tot el que ve després.

- La cua de preparats no és «els processos que hi ha», sinó els que tenen una ràfega començada o a punt de començar. Un procés bloquejat no hi és perquè en aquell moment no en té cap.
- El quantum no és «quant dura un procés»: és fins on deixem arribar una ràfega abans de tallar-la.

- I quan comparem dues polítiques, el que comparem és com ordenen ràfegues.

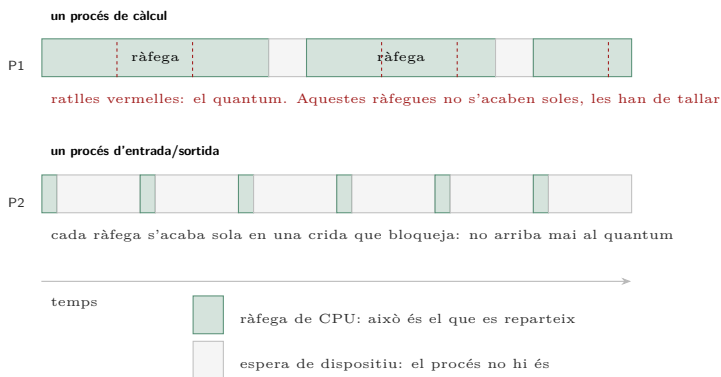


Figura 6.1. Els mateixos dos processos a la mateixa escala de temps.

6.2.1 Dues maneres de viure

Gairebé tots els processos alternen ràfegues i expectes, però no en la mateixa proporció, i la diferència no és de naturalesa sinó de mida:

- Un procés **de càlcul** fa poques ràfegues i llarguíssimes. Sovint només en fa una, que dura tot el programa, i que el quantum va tallant a trossos. Una compilació, un vídeo comprimint-se.
- Un procés **d'entrada/sortida** en fa moltíssimes i minúscules. Un editor de text: entre tecla i tecla no fa res, i quan en premeu una calcula uns quants microsegons i torna a esperar.

Compte

«De càlcul» i «d'entrada/sortida» no són etiquetes que el procés dugui enganxades: són descripcions de com són les seves ràfegues, i poden canviar mentre s'executa. El mateix compilador és un procés d'entrada/sortida mentre llegeix els fitxers i un procés de càlcul mentre optimitza. El planificador no ho sap per endavant,

i no li cal: només ha de mirar què ha passat fins ara.

6.2.2 🌀 Mirar-se les ràfegues

Tot això es pot comprovar sense entrar al kernel, perquè el kernel en porta el compte i el deixa mirar: quantes vegades un procés ha deixat la CPU perquè ha volgut, i quantes l'hi han pres. La suma és quants torns de CPU ha tingut.

Llistat 6.1. plani-rafegues: quants torns i de quina mida

```

1  int main(int argc, char *argv[])
2  {
3      double cpu, torns;
4      struct rusage r;
5      char buf[192];
6      long n;
7
8      if (argc != 3 || (n = atol(argv[2])) < 1)
9          Usage(argv[0]);
10     if (strcmp(argv[1], "calcul") == 0)
11         calcula(n);
12     else if (strcmp(argv[1], "es") == 0)
13         espera(n);
14     else
15         Usage(argv[0]);
16
17     getrusage(RUSAGE_SELF, &r);
18     cpu = r.ru_etime.tv_sec + r.ru_etime.tv_usec / 1e6
19         + r.ru_stime.tv_sec + r.ru_stime.tv_usec / 1e6;
20     torns = r.ru_nvcsw + r.runivcsw;
21
22     sprintf(buf,
23             "%s: %.3f s de CPU\n"
24             "  la deixa ell %ld cops, l'hi prenen %ld\n"
25             "  CPU per torn: %.3f ms\n",
26             argv[1], cpu, r.ru_nvcsw, r.runivcsw,
27             torns ? cpu * 1000 / torns : 0);
28     write(1, buf, strlen(buf));
29     return 0;
30 }
```

- ❶ El `getrusage()` dóna la comptabilitat que el kernel ja porta d'aquest procés; no cal instrumentar res.
- ❷ `ru_nvcsw` són els canvis de context *voluntaris* — la CPU l'ha deixada ell — i `ru_nivcsw` els *involuntaris* — l'hi han pres.
- ❸ I la divisió: quanta CPU li ha tocat cada vegada que l'ha tinguda.

```
$ ./plani-rafegues calcul 2
calcul: 2.002 s de CPU
  la deixa ell 0 cops, l'hi prenen 3
  CPU per torn: 667.273 ms
$ ./plani-rafegues es 2000
es: 0.019 s de CPU
  la deixa ell 2001 cops, l'hi prenen 0
  CPU per torn: 0.010 ms
```

Mireu les dues columnes del mig abans que els temps. El procés de càlcul **no deixa la CPU ni una sola vegada**: no s'hi bloqueja mai, i les tres vegades que la perd és perquè l'hi prenen. El d'entrada/sortida és el contrari exacte: la deixa dues mil vegades i no cal prendre-li mai, perquè cada ràfega s'acaba sola molt abans que el quantum.

I ara sí, els temps: 667 mil·lisegons contra 0,010. En una altra màquina del laboratori surten vuit tornos i 250 mil·lisegons: quants us en toquin depèn de la màquina i de qui més hi ha corrent. La proporció, en canvi, no es mou gaire. Gairebé cinc ordres de magnitud, entre dos programes que corren a la mateixa màquina i el mateix segon.

Compte

Els 667 mil·lisegons del procés de càlcul no són la mida de la seva ràfega: la seva ràfega dura els dos segons sencers. Són la mida del *tros* que el sistema li deixa fer seguit, i en una màquina buida com aquesta li'n deixa fer de molt llargs. Proveu-ho amb tres còpies alhora i mireu com baixa.

La conseqüència de disseny és la que explica el capítol sencer: en un sistema real, la immensa majoria de ràfegues són curtes. Per això

un quantum de desenes de mil·lisegons quasi mai no s'esgota, i per això l'expropiació — que semblava el cor de tot plegat — acaba sent el cas rar.

6.3 Les cues del kernel

Del PCB ja en sabeu prou: al capítol 5 vam veure què hi ha a dins. El que falta és saber com estan organitzats, perquè tenir mil PCB en un munt no serveix de res: el planificador ha de poder respondre en microsegons a la pregunta «qui és el següent».

La resposta habitual són cues. Hi ha una llista amb tots els processos del sistema, i després n'hi ha una per a cada motiu pel qual un procés pot estar esperant: la **cua de ready** amb els que volen CPU*, i una **cua per dispositiu** amb els que esperen que el disc, la xarxa o el terminal els contesti.

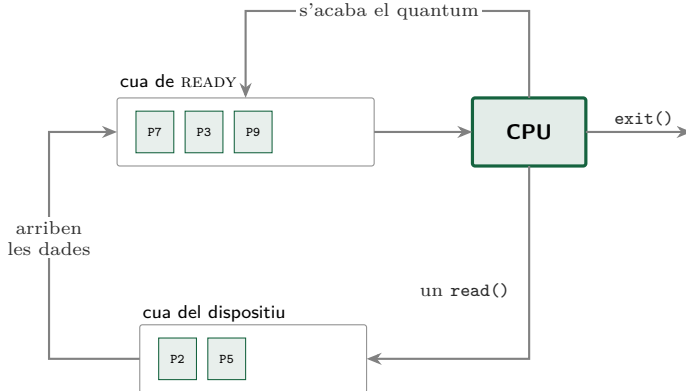


Figura 6.2. Un procés no travessa el planificador: hi dóna voltes. Cada fletxa d'aquest dibuix és una fletxa del graf d'estats del capítol 5.

Comparant aquesta figura amb el graf d'estats de la pàgina 74 es veu que són el mateix dibuix dues vegades: allà, els estats d'un procés;

*A la pràctica no n'és una sola. Els sistemes reals les separen per prioritat, per classe de procés o per CPU, i a Linux ni tan sols és una cua: és un arbre. Ho veurem al final.

aquí, les estructures on el kernel guarda els processos que hi són. Un procés no «està en estat READY» en cap camp misteriós: està a la cua de READY, i canviar-lo d'estat és treure'l d'una llista i posar-lo en una altra.

De triar l'estructura, en depèn més del que sembla. Cal que inserir i treure siguin ràpids, perquè es fa a cada canvi de context; cal que buscar-hi sigui ràpid, i cal decidir per quin camp es busca — pel PID? per l'usuari?; i cal que tot plegat aguantí quan a la màquina no hi ha deu processos sinó deu mil.

6.4 El canvi de context

Quan un procés deixa la CPU i n'hi entra un altre, el kernel ha de desar tot allò que fa que el primer pugui continuar més tard, i restaurar el mateix del segon. Això és un **canvi de context**.

El que es desa és el context de maquinari del PCB: els registres, el comptador de programa, l'estat de la unitat de coma flotant, el que faci falta perquè el procés no s'assabenti de res quan torni. I el lloc on es desa és el PCB mateix, que per això té un forat de la mida justa.

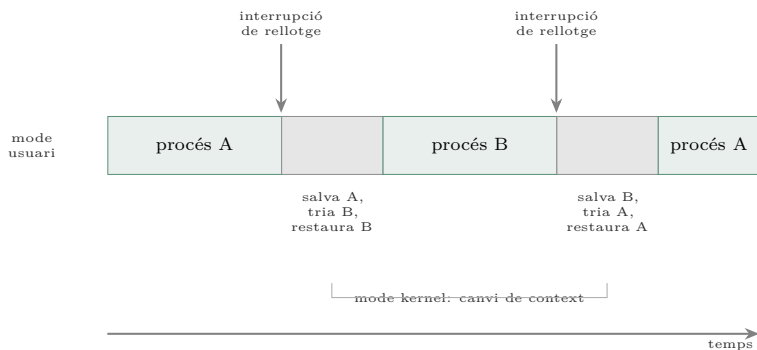


Figura 6.3. Els trossos grisos són el canvi de context. No fan avançar cap programa: per això han de ser curts.

El canvi de context no és temps útil. Mentre dura, cap programa de l'usuari no avança ni una instrucció: la màquina està ocupada recordant on érem. Tot el que es discuteix en aquest capítol sobre quantums i freqüències surt, al capdavant, de voler que aquesta estona sigui una fracció petita de la resta.

Per això els processadors hi ajuden. Moltes arquitectures tenen instruccions que salven o restauren un bloc sencer de registres d'un cop, i alguns sistemes eviten desar l'estat de la coma flotant fins que el procés nou el toca de debò — si no el toca mai, la feina no s'ha fet.

6.5 Quan s'executa el planificador

El planificador no s'està executant contínuament: és codi del kernel, i el kernel només s'executa quan alguna cosa el crida. Les ocasions són les dues maneres d'acabar una ràfega que ja hem vist, i ara tenen nom.

El procés que té la CPU no pot continuar. Ha fet un `exit()`, o s'ha bloquejat en un `read()` o un `waitpid()`. Aquí no hi ha res a decidir sobre si se li treu la CPU — ja l'ha deixada ell —, només cal triar el següent. Aquests esdeveniments es diuen **no apropiatius** (*non-preemptive*).

El procés que té la CPU podria continuar, però se li treu. S'ha acabat el seu torn, o acaba d'aparèixer un procés que el sistema considera més urgent. Aquests són els **apropiatius** (*preemptive*), i són els que demanen que el kernel pugui recuperar el control d'un procés que no pensava tornar-lo.

Una política que només tracta esdeveniments del primer tipus és **no apropiativa**: mai no treu la CPU a ningú, i espera que el procés l'alliberi. Una que també tracta els del segon és **apropiativa**, i necessita, sí o sí, una interrupció de rellotge.

Aquella interrupció és el que fa possible tot plegat, i és la resposta

a la qüestió que us vam deixar plantejada al capítol 5: cada pocs mil·lisegons el maquinari interromp el que s'estigui executant i passa el control al kernel, que aprofita per mirar si cal canviar de procés. Sense ella, un bucle infinit en un programa d'usuari deixaria la màquina penjada per sempre — que és exactament el que passava als sistemes que no en tenien.

6.6 Què volem optimitzar

Abans de comparar polítiques cal saber què es mesura, perquè si no, «millor» no vol dir res. Les mesures habituals són aquestes:

Temps de retorn (*turnaround time*) el temps total des que el procés arriba al sistema fins que acaba. Hi entra tot: la CPU que ha consumit, el que ha esperat a la cua de READY i el que ha esperat el disc.

Temps d'espera només el que ha passat a la cua de READY. És la part que el planificador pot millorar; les altres no depenen d'ell.

Temps de resposta el que passa des que li demaneu una cosa fins que en veieu la primera senyal. És el que nota la persona que és davant de la pantalla, i sovint importa més que els altres dos.

Productivitat (*throughput*) quants processos acaba el sistema per unitat de temps.

Compte

Cap política no les optimitza totes alhora, i triar-ne una és renunciar a alguna cosa. Un sistema que vol el millor temps de resposta canviarà de procés molt sovint, i cada canvi és temps perdut, de manera que la productivitat baixarà. Quan compareu dues polítiques, la primera pregunta no és quina és millor: és millor *per a què*.

I hi ha una segona cosa que cal tenir present, que ja sabeu: que les ràfegues dels processos de càlcul i les dels d'entrada/sortida es

diferencien en quatre ordres de magnitud. Torna d'aquí a dues seccions, perquè és exactament on el Round Robin ensopega.

6.7 Round Robin

És la política més simple que funciona, i és per això que l'expliquem encara que cap sistema actual no la faci servir tal qual.

La regla és una sola: els processos preparats fan cua per ordre d'arribada, i cadascun rep la CPU durant un temps màxim fix, el **quantum**. Si l'esgota, se li treu i va al final de la cua; si abans es bloqueja o acaba, se'n va i entra el següent.

Els tres esdeveniments que la posen en marxa són, doncs, els del capítol: el procés acaba i el procés es bloqueja — no apropiatius — i s'acaba el quantum, que sí que ho és. Round Robin és, per tant, una política apropiativa, i necessita el rellotge.

Vegem-la amb tres processos que arriben alhora i que demanen 5, 3 i 4 unitats de CPU, amb un quantum de 2:

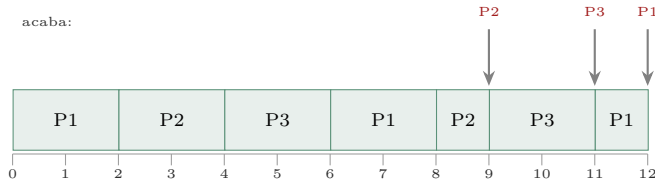


Figura 6.4. Round Robin amb quantum 2. Ningú no fa més de dues unitats seguides mentre hi hagi algú esperant.

	Demana	Acaba a	Ha esperat
P1	5	12	7
P2	3	9	6
P3	4	11	7
Mitjana		10,7	6,7

Fixeu-vos que el P2, que era el més curt, és el que acaba primer, i que això no és cap decisió de la política: surt sol. I fixeu-vos també que tots tres han esperat molt en relació amb el que demanaven. Round Robin no és bo en temps de retorn; el que és, és *previsible*.

6.7.1 El quantum

Tota la política es juga en un sol número, i les dues respostes fàcils són totes dues dolentes.

Si el quantum és molt **gran**, ningú no l'esgota mai: cada procés tindrà la CPU fins que es bloquegi o acabi, i Round Robin es converteix en «el primer que arriba, el primer que se serveix». Un procés llarg deixa la resta de la màquina esperant.

Si és molt **petit**, els processos s'alternen molt de pressa i el temps de resposta és boníssim, però entre torn i torn hi ha un canvi de context. Amb un quantum de la mateixa mida que el canvi de context, la meitat de la màquina se'n va a recordar on érem.

El quantum ha de ser **gran comparat amb el canvi de context i petit comparat amb el que la gent nota**. Entre aquestes dues fites hi ha uns quants ordres de magnitud, i per això els valors habituals — entre 10 i 100 mil·lisegons — funcionen prou bé sense haver-los d'afinar.

Amb N processos a la cua i un quantum Q , cada procés rep una part d' N del temps de CPU, en trossos de com a molt Q , i cap no espera més de $(N - 1)Q$ per tornar a tenir-ne. Aquesta última és la propietat que fa que Round Robin s'ensenyi encara: dóna una garantia, i és una garantia que es pot calcular.

6.8 El que el Round Robin fa malament

Torneu a la figura 6.1 i poseu les dues tires a la mateixa cua.

El procés de càlcul té una ràfega més llarga que el quantum, o sigui que l'esgota sencer cada vegada que li toca. El d'entrada/sortida té ràfegues de microsegons: fa quatre coses, es bloqueja esperant una tecla o el disc, i torna la CPU al cap de la mil·lèsima part del que li havien concedit. Ara mireu-vos què passa quan tots dos fan cua junts.

i Compte

Round Robin dóna a tothom el mateix *torn*, no la mateixa *CPU*. El procés de càlcul se n'endú el quantum sencer cada volta; el d'E/S, una engruna. Al cap d'un segon, el que s'ho ha quedat gairebé tot és el que ja en tenia prou, i l'editor de text va a batzegades.

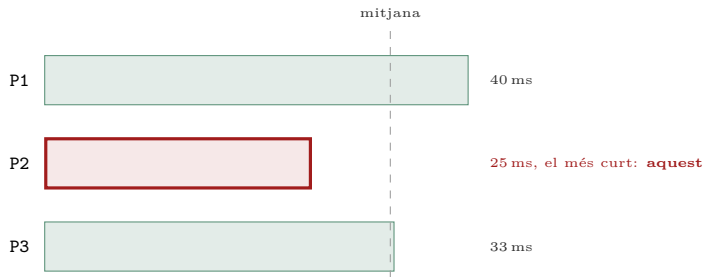
El problema no és que la política sigui injusta per descuit: és que fa una cosa diferent de la que crèiem. Repartir torns i repartir temps són coses diferents, i el que volem és el segon.

6.9 Completely Fair Scheduling

Aquesta és la idea que Linux fa servir des del 2007*, i el gir és petit i radical alhora: en comptes de repartir torns, portar el compte del temps de CPU que cadascú ha consumit.

El CFS es pregunta, per a cada procés, quanta CPU *hauria* de tenir si el repartiment fos perfectament just, i quanta en té de debò. Executa sempre el que va més endarrerit respecte del seu ideal.

*Des del kernel 2.6.23, i amb el nom d'*scheduler* completament just. Els kernels d'ara ja no el porten: vegeu el requadre al final de la secció.



Cada barra és la CPU que aquell procés ha consumit fins ara. No hi entren ni torns ni prioritats: es tria sempre la més curta, i així les tres tendeixen a igualar-se.

Figura 6.5. La decisió del CFS: la barra més curta.

D'això en surten unes quantes conseqüències.

No hi ha quantum fix. El temps màxim que un procés pot estar seguit a la CPU es calcula: depèn de quants n'hi ha competint i de com de lluny està cadascun del seu ideal. Amb pocs processos, els trossos són llargs i es canvia poc; amb molts, s'escurcen.

Els processos d'E/S deixen de sortir perjudicats, i no perquè se'ls tracti bé: perquè el comptador diu la veritat. Un editor de text que ha consumit dos mil·lisegons de CPU en tot un segon va molt endarrerit respecte del seu ideal, i per tant, quan prem una tecla, és el primer de la llista. La resposta instantània que noteu no és una prioritat especial: és aritmètica.

La cua deixa de ser una cua. Si el que s'ha de fer és «tria el que té el comptador més petit», una llista per ordre d'arribada no serveix. El CFS fa servir un arbre equilibrat ordenat per aquell comptador, de manera que triar el següent és anar a l'element de més a l'esquerra.

I es pot comptar per grups. L'administrador pot dir que tots els processos d'un usuari, o d'un servei, comptin com un de sol a l'hora de repartir. Així, un usuari que llanci cent processos no s'endú cent parts de la màquina: se n'endú una, i se la reparteix entre els seus.

▲ Qüestió 1

Amb Round Robin, obrir cent finestres d'un programa em dona cent vegades més CPU que al veí que només n'ha obert una. Amb el CFS per grups, no. Quina de les dues coses és «justa»?

💡 Resposta

Totes dues, i aquest és el punt. La primera és justa entre *processos*; la segona, entre *usuaris*. Cap política no pot decidir tota sola quina de les dues justícies volem: és una decisió de qui administra la màquina, i per això el CFS la deixa configurable en comptes de triar-la ell. Quan llegiu que un planificador és «just», la pregunta útil és sempre la mateixa: just entre qui i qui.

🔍 Per saber-ne més

Des del kernel 6.6, del 2023, el planificador de Linux ja no és el CFS sinó l'EEVDF (*Earliest Eligible Virtual Deadline First*), i les màquines del laboratori el fan servir. El que canvia no és la idea de fons — portar el compte del temps consumit i afavorir qui va endarrerit —, sinó com es desfà l'empat: a cada procés se li posa un termini virtual, calculat a partir del temps que li tocava, i s'executa el que el té més a prop. Amb això, l'estona que un procés interactiu pot arribar a esperar queda acotada, cosa que el CFS no prometia enlloc. Tot el que hem explicat en aquesta secció segueix valent; la lletra petita és a [Documentation/scheduler/](#) del codi del kernel.

6.10 Mirar-s'ho des de fora

Res de tot això no es veu des d'un programa, però se'n poden veure els efectes. Aquest fa voltes durant tres segons de rellotge de paret i després diu quanta CPU li han donat de debò:

Llistat 6.2. Quanta CPU m'ha tocat

```

1  int main(void)
2  {
3      char buf[96];
4      time_t inici;
5      clock_t cpu;
6      double consumida;
7      long voltes = 0;
8
9      inici = time(NULL);
10     cpu = clock();
11
12     while (time(NULL) - inici < SEGONS)
13         voltes++;
14
15     consumida = (double)(clock() - cpu) / CLOCKS_PER_SEC;
16     sprintf(buf, "pid %d: %ld voltes, %.2f s de CPU\n",
17             (int) getpid(), voltes, consumida);
18     write(1, buf, strlen(buf));
19     return 0;
20 }

```

- ❶ `time()` dóna el temps de paret; `clock()`, el de CPU consumida per aquest procés. Que siguin dues coses diferents és el tema del capítol.
- ❷ El bucle no fa res útil a posta: només vol la CPU.
- ❸ I la diferència entre tots dos és el que us diu quant heu hagut de compartir.

Executeu-lo sol i després unes quantes còpies alhora. Compte amb una cosa: si la màquina té diversos cores, quatre còpies s'executaran de debò en paral·lel i no compartiran res. Per veure el repartiment cal obligar-les a fer cua en un sol core, i això és el que fa el `taskset`:

```

alumne@labS0:~/S3$ ./plani-repartiment
pid 8169: 1399805598 voltes, 2.56 s de CPU
alumne@labS0:~/S3$ for i in 1 2 3 4; do
>   taskset -c 0 ./plani-repartiment &
> done; wait
pid 8172: 398232609 voltes, 0.75 s de CPU
pid 8175: 397429357 voltes, 0.75 s de CPU
pid 8174: 400544049 voltes, 0.75 s de CPU

```

```
pid 8173: 398865396 voltes, 0.75 s de CPU
```

Tres segons de paret en tots dos casos. Sol, el procés se n'endú gairebé tota la CPU d'aquell core; amb quatre competint, cadascú se n'endú just una quarta part, amb una regularitat que fa una mica d'angúnia.

Compte

El `-c 0` del `taskset` no vol dir «el core zero»: vol dir «la CPU número zero», i **una CPU de Linux no és un core**. Si la vostra màquina té SMT —dos jocs de registres per core, que Intel en diu *hyper-threading*—, el `lscpu` us dirà que teniu vuit CPU quan en realitat teniu quatre cores, i dues d'aquelles vuit compartiran maquinari sense que el número ho digui.

Pitjor encara: **quines dues es reparteixen el mateix core depèn de la màquina**, i no us ho podeu inventar. Ho heu de preguntar. Aquesta és una de les màquines del laboratori:

```
$ lscpu -e=CPU,CORE,SOCKET
CPU CORE SOCKET
0      0      0
1      1      0
2      2      0
3      3      0
4      4      0
5      5      0
6      0      0
7      1      0
8      2      0
9      3      0
10     4      0
11     5      0
```

Dotze CPU i sis cores: les sis primeres són un joc de registres

de cada core, i les sis següents, l'altre joc. La CPU 0 i la 6 són el mateix core, la 1 i la 7 també, i així fins al final. O sigui que al laboratori un `taskset -c 0,1` posa les dues còpies en cores diferents, i un `taskset -c 0,6` les posa al mateix —i la mesura que féssiu no voldria dir el que us penseu. I no és cap regla: en altres aules de la mateixa facultat la germana de la CPU 0 és la 1. Mireu-vos la taula abans de fixar res.

I encara una altra: en moltes màquines d'avui **els cores no són tots iguals** —n'hi ha de ràpids i d'estalviadors—, de manera que fixar una còpia al zero i una altra al quatre pot donar temps diferents sense que hi hagi cap misteri. Al capítol 12 hi ha com mirar-ho.

Això és el planificador fent la seva feina, i és tot el que en veureu des de fora.

Per saber-ne més

Al repartiment s'hi pot influir des de fora, i el dial es diu *nice*: un número de -20 a 19 que diu com d'amable esteu disposat a ser. Com més alt, menys pes al repartiment i més CPU per als altres. Pujar-lo el pot fer qualsevol; baixar-lo, només el *root*, i el motiu és evident.

```
$ strace -qq -e nice,setpriority nice -n 19
./plani-repartiment
setpriority(PRIO_PROCESS, 0, 19)      = 0
pid 8342: 1660623637 voltes, 2.99 s de CPU
```

Fixeu-vos en el nom que surt a la primera línia, que no és el que heu escrit. La comanda es diu **nice**, la funció de llibreria es diu **nice()**, i el que arriba al kernel es diu **setpriority()**. De fet el **nice()** no existeix com a crida a cap de les dues màquines d'aquest llibre: la libc el fa veure. L'apèndix A explica per què passa això i com comprovar-ho a la vostra.

6.11 Exercicis

1. Refeu la traça de la figura 6.4 amb un quantum de 1 i amb un quantum de 5. Com canvien el temps de retorn mitjà i el nombre de canvis de context? Quin dels dos quantums triaríeu, i per a quina mena de màquina?
2. Als mateixos tres processos, apliqueu-hi una política que no sigui apropiativa: cadascun té la CPU fins que acaba. Surt un temps de retorn mitjà més bo o més dolent? I si el P1 hagués demanat 50 en comptes de 5?
3. Executeu el `plani-repartiment` amb vuit còpies sobre un sol core. Quanta CPU li toca a cadascuna? I si a una de les vuit li poseu un `nice -n 19` al davant?
4. Compareu el temps de CPU que consumeix el `plani-repartiment` sol amb el que consumeix mentre a la mateixa màquina hi teniu una compilació en marxa. El vostre programa és de càlcul o d'entrada/sortida? I la compilació?
5. Amb `cat /proc/self/status` podeu veure el vostre propi PCB, o el tros que el kernel us en deixa veure. Busqueu-hi els camps `voluntary_ctxt_switches` i `nonvoluntary_ctxt_switches`: són els dos números que el `plani-rafogues` demana amb el `getrusage()`. Quin dels dos correspon als esdeveniments apropiatius?
6. Executeu tres còpies del `plani-rafogues` calcul 5 alhora i compareu la CPU per torn amb la d'una sola còpia. Ara feu-ho amb `taskset -c 0`, que les posa totes en un sol core. Quin dels dos números de la comptabilitat canvia, i per què?

Bibliografia

- [1] Abraham Silberschatz, Peter B. Galvin i Greg Gagne. *Operating System Concepts*. 10a ed. Hoboken, NJ: Wiley, 2018. ISBN: 978-1-118-06333-0.