Université Claude Bernard Lyon 1

MIF18 - Systèmes avancés

TP 2 - Ordonnancement Linux

Repo GIT
https://github.com/d4wae89d498/mif18-labs23/tree/master/TP_Ordo_Linux/_code

Marc FAUSSURIER
p1707031

Sommaire

Caractéristiques de la machine

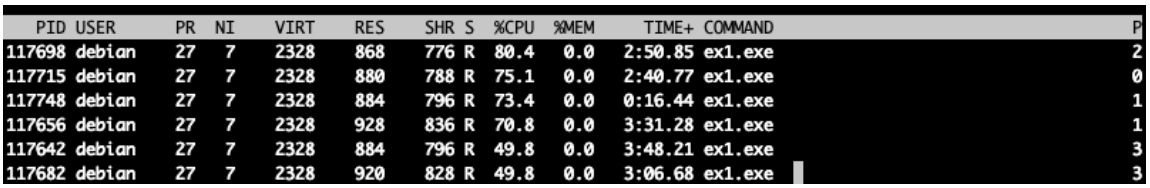
La machine sur laquelle les tests seront effectués est un VPS disposant de 4 cœurs et 4 GO de RAM. La distribution est Debian 12.

```
debian@vps-11d5cfd5:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 25
model           : 1
model name      : AMD EPYC-Milan Processor
stepping        : 1
microcode       : 0x1000065
cpu MHz         : 2295.684
cache size      : 512 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 syscall
nx mmxext fxsr_opt pdpe1gb rdtscp lm rep_good nopl cpuid extd_apicid tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid sse4_1
sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm svm cr8_legacy abm sse4a misalignsse 3dnowprefetch
osvw topoext perfctr_core invpcid_single ssbd ibrs ibpb stibp vmmcall fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap
clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves clzero xsaveerptr wbnoinvd arat npt nrip_save umip pku ospke rdpid fsrm
bugs            : sysret_ss_attrs null_seg spectre_v1 spectre_v2 spec_store_bypass srso
bogomips        : 4591.36
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
....
```

Premières observations

Écrivez dans ex1.c un programme qui fait une attente active infinie avec while(1).

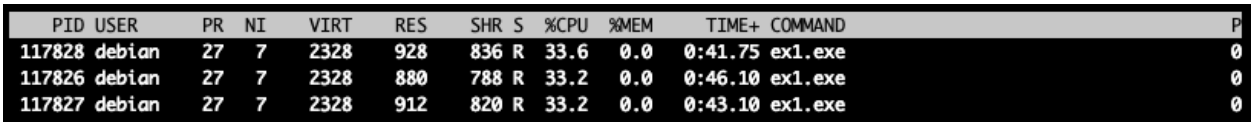
Que pouvez-vous observer dans top quand vous lancez plusieurs fois ce programme (sur des terminaux différents) ?



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND	P
117698	debian	27	7	2328	868	776	R	80.4	0.0	2:50.85	ex1.exe	2
117715	debian	27	7	2328	880	788	R	75.1	0.0	2:40.77	ex1.exe	0
117748	debian	27	7	2328	884	796	R	73.4	0.0	0:16.44	ex1.exe	1
117656	debian	27	7	2328	928	836	R	70.8	0.0	3:31.28	ex1.exe	1
117642	debian	27	7	2328	884	796	R	49.8	0.0	3:48.21	ex1.exe	3
117682	debian	27	7	2328	920	828	R	49.8	0.0	3:06.68	ex1.exe	3

J'observe que le système d'exploitation Linux tente d'assigner chaque instance du programme exécuté à un cœur de processeur distinct. Lorsque tous les cœurs sont utilisés, l'utilisation du CPU se répartit entre les différents processus.

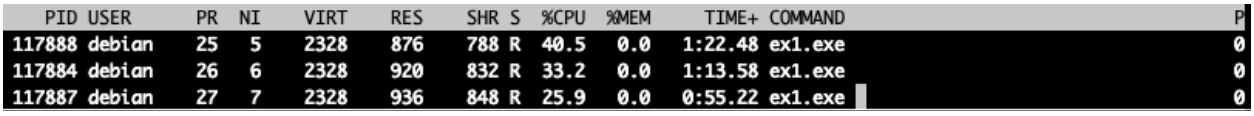
Si vous bindez les différentes instances du programme sur un seul coeur, comment se répartit le temps processeur ?



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND	P
117828	debian	27	7	2328	928	836	R	33.6	0.0	0:41.75	ex1.exe	0
117826	debian	27	7	2328	880	788	R	33.2	0.0	0:46.10	ex1.exe	0
117827	debian	27	7	2328	912	820	R	33.2	0.0	0:43.10	ex1.exe	0

J’observe que le temps processeur se reparti équitablement entre les différentes instances du programme.

Jouez maintenant avec la priorité en ayant plusieurs instances de votre programme avec des priorités différentes. Qu'observez-vous ?



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND	P
117888	debian	25	5	2328	876	788	R	40.5	0.0	1:22.48	ex1.exe	0
117884	debian	26	6	2328	920	832	R	33.2	0.0	1:13.58	ex1.exe	0
117887	debian	27	7	2328	936	848	R	25.9	0.0	0:55.22	ex1.exe	0

J’observe que plus la priorité est basse ; plus du temps processeur est alloué à l’instance du programme.

Mesure des changements de contexte

Le fichier ex2.c contient un code qui permet de mesurer le temps passé dans une boucle.

Écrivez un programme qui effectue N=100 000 appels à sched_yield() puis affiche le temps requis pour la somme de ces appels et recommence.

Si vous lancez deux instances du programme (sur un même processeur), que signifie le temps affiché? Quel est son sens avec une seule instance du programme ?

Avec une instance du programme

```
debian@vps-11d5cfd5:~/lyon1/mif18-labs23/TP_Ordo_Linux/_code$ ./ex2.exe
Calling 100000 sched_yield() took 34369 microseconds
Calling 100000 sched_yield() took 35798 microseconds
Calling 100000 sched_yield() took 30570 microseconds
Calling 100000 sched_yield() took 34422 microseconds
Calling 100000 sched_yield() took 31420 microseconds
Calling 100000 sched_yield() took 32578 microseconds
```

Avec deux instances du programme sur le même cœur

```
651 Calling 100000 sched_yield() took 186550 microseconds
652 Calling 100000 sched_yield() took 169992 microseconds
653 Calling 100000 sched_yield() took 180135 microseconds
654 Calling 100000 sched_yield() took 183320 microseconds
655
term://~/lyon1/mif18-labs23/TP_Ordo_Linux/_code//123028:/bin/bash
537 Calling 100000 sched_yield() took 186282 microseconds
538 Calling 100000 sched_yield() took 182282 microseconds
539 Calling 100000 sched_yield() took 176062 microseconds
540 Calling 100000 sched_yield() took 184977 microseconds
```

Lorsqu'on lance deux instances du programme sur un même processeur, le temps affiché représente la durée totale nécessaire pour effectuer que le processus soit réélu 100 000 par l'ordonnanceur pour chaque instance du programme, ce temps est influencé par l'ordonnancement du système d'exploitation et la concurrence des ressources CPU entre les processus. Le temps est donc un indicateur de l'efficacité de l'ordonnanceur et de la surcharge due aux changements de contexte.

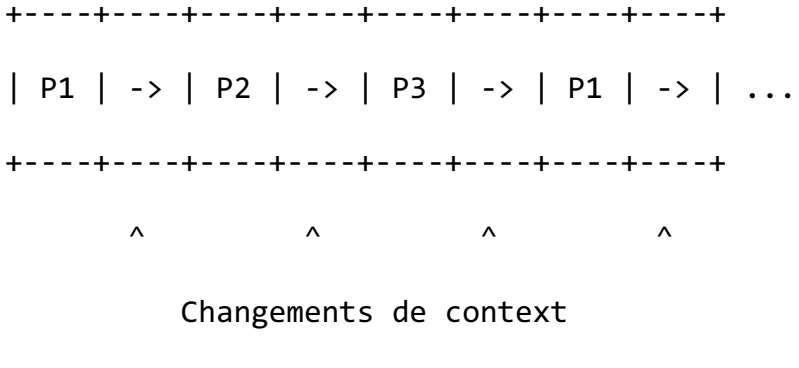
Avec une seule instance, le temps reflète la surcharge inhérente à l'exécution de 100 000 appels à sched_yield() sans concurrence directe. D'une manière simplifiée ; cela mesure le coût de l'opération de réélection pour le processus en question dans un environnement isolé où l'ordonnanceur n'a pas à alterner entre plusieurs tâches concurrentes pour le même cœur de processeur.

Complétez un tableau avec le temps entre deux sched_yield() avec 1, 2, 3, 4 et 5 instances du programme. Prenez bien garde de prendre un temps cohérent uniquement une fois que toutes les instances du programme ont bien été lancées. Vous pouvez par exemple les lancer en tâche de fond et prendre le temps après 10 tours de boucle qui fait les N=100 000 appels à sched_yield().

Nombre d'instance du programme	Moyenne des 10 mesures de N appels en us (M)	*us entre deux sched_yield* (M/N)
1	33225	0,33225
2	185588	1,85588
3	287950	2,87950
4	381385	3,81385
5	476394	4,76394

Commentez. Quel est le temps d'un changement de contexte avec une seule instance du programme ? Avec plusieurs instances ? Faites un dessin pour expliquer ce qu'il se passe avec trois instances du programme.

Avec une instance, on a une moyenne de 0,33225 ms pour un changement de contexte. Quand on augmente le nombre d'instances, le temps entre deux sched_yield() s'accroît car le système d'exploitation doit jongler entre plus de processus, donc plus de changements de contexte sont nécessaires avant que le processus soit réélu par l'ordonnanceur.



Comment le coût d'un changement de contexte va t-il jouer sur les performances en fonction de la timeslice, la durée durant laquelle le processus tourne sans être interrompu par un autre processus ?

Le coût d'un changement de contexte peut réduire les performances lorsque la timeslice est trop courte, car le temps passé à changer de contexte peut devenir significatif par rapport à l'exécution effective du processus. À l'inverse, une timeslice plus longue réduit la fréquence des changements de contexte mais peut augmenter la latence de réponse pour d'autres processus, affectant l'équité et la réactivité du système. Un équilibre doit donc être trouvé par l'ordonnanceur afin d'optimiser le débit global et le temps de réponse.

Mesure des timeslices

Écrivez un programme ex3.c. qui fait des gettimeofday() et qui affiche un message lorsque la durée entre deux appels est inhabituellement longue. Cela signifiera alors que le programme a été désordonnancé. Le message devra contenir cette durée afin d'avoir une estimation de la timeslice*.*

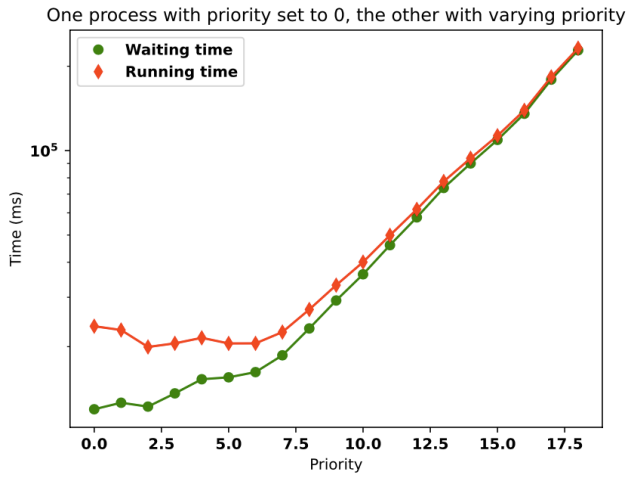
Afin de limiter le bruit (issu des autres processus qui s'exécutent sur votre machine), vous n'afficherez le message que lorsque la durée entre deux appels sera suffisante (par exemple 0.5 ms ou 1 ms).

Expliquez le comportement observé quand vous lancez une ou deux instances de ce programme (toujours sur le même processeur).

Tracez un graphe montrant l'évolution de la timeslice*en fonction de la priorité. Avec deux instances du programme, vous ne jouerez qu'avec la priorité d'une des instances. Le code fourni dans le programme python analyze_ex3.py permet via la fonction acquire_results() de lancer deux instances du programme dont une avec une priorité variable (de 0 à 19). Après une attente de 20 secondes, les deux processus sont tués. Il vous est demandé de compléter ce code afin de parser les fichiers générés par votre executable ex3. Vous utiliserez python3 analyze_ex3.py pour lancer les expériences (cela prendra du temps) et python3 analyze_ex3.py parse pour parser et afficher les données.*

Qu'en concluez vous ?

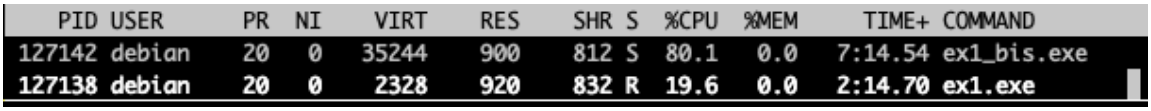
Nous avons pu observer que la timeslice fluctue de manière significative lorsqu'un processus est désordonnancé, ce qui indique une interférence notable avec d'autres processus. La variation des priorités a démontré que plus la priorité d'une instance est élevée, plus la timeslice tend à être stable. Le graphe généré illustre cette tendance. Cette expérimentation souligne l'importance de la gestion des priorités dans la programmation concurrente.



Thread vs processus

Comparez (en bindant sur le même coeur) un processus mono-threadé qui fait une boucle infinie et un processus multi-threadé où chaque thread fait une boucle infinie. Comment se répartissent-ils le temps CPU ?

Afin de répondre à cette question un programme ex1_bis.c a été créé pour effectuer une boucle infinie dans des threads. Ce programme a été lancé avec 4 threads et a été mis en concurrence sur un cœur avec ex1.c, qui fait lui une simple boucle infinie dans un processus sans multi-threading.



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
127142	debian	20	0	35244	900	812	S	80.1	0.0	7:14.54	ex1_bis.exe
127138	debian	20	0	2328	920	832	R	19.6	0.0	2:14.70	ex1.exe

On remarque que le processus ex1_bis.exe occupe 4 fois plus souvent le processeur.

Vous pouvez partir du fichier ex2_bis.c afin d'avoir une base de code qui créé les threads pour vous.

Mesurez les changements de contexte. Que se passe t-il ? Vous pouvez vérifier qui a la main après chaque appel à sched_yield() en ajoutant de l'affichage. Attention de bien l'enlever quand vous voulez faire des mesures de temps.

Je remarque que les résultats pour N=100000 appels à sched_yield() et NB threads ont une évolution similaire à NB instances de l'exercice 2. Cependant les moyennes sont plus basses pour un même nombre de threads que que nombre d'instance d'un programme. En effet, en multithrading, l'appel système sched_yield va faire permutter le thread qui occupe la place dans l'ordonnanceur du système d'exploitation, les threads d'un même processus étants plus légers que deux instances d'un même programme ; le changement de contexte a un cout plus faible. Pour effectuer les mesures du tableau ci-dessous ; j'ai utilisé une barrière pthread `pthread_barrier_t` dans ex2_bis.c afin de démarrer tous les threads en même temps. J'ai également utilisé la fonction `clock_gettime` afin d'essayer d'être le plus précis possible dans mes mesures.

Nombre de threads	Moyenne des 20 mesures de N appels en us (M)	*us entre deux sched_yield* (M/N)
1	33506	0,33506
2	35193	0,35193
3	38548	0,38548
4	39390	0,39390
5	60422	0,60422
10	210167	2,10167

Conclusion

L'ordonnancement dans Linux assure que chaque processus reçoit du temps de processeur de manière équitable tout en respectant les priorités définies, permettant ainsi le fonctionnement multitâche. Les changements de contexte sont des opérations coûteuses car ces changements nécessitent de sauvegarder et de charger les états des processus, ce qui peut impacter les performances lorsqu'ils sont trop fréquents.

Le multithreading est une alternative pour un programme qui est plus légère que le multi-processing pour faire la programmation concurrente ; en effet les threads partageant beaucoup de données entre eux ; les changements de contexte sont donc beaucoup plus courts. Les barrières de sécurité entre les threads d'un même processus sont plus limitées, en cas de bugs mais aussi en cas d'exécution de code malveillant.

De nouveaux systèmes de programmation concurrentes ont également été développés récemment :

- Les goroutines de Go, légères et gérées par le runtime. Elles utilisent un modèle de communication via des canaux.
- Les fibers en C/C++ sont des threads légers gérés par le développeur. Elles permettent un contrôle fin sur la planification, mais nécessitent une gestion manuelle des ressources.
- Les coroutines de C++ 20 offrent un moyen de suspendre et de reprendre l'exécution de fonctions, facilitant la programmation asynchrone et la gestion des tâches bloquantes.

Il serait intéressant de poursuivre sur ces nouveaux mécanismes afin d'avoir une vision d'ensemble ; tout comme il serait intéressant de construire un ordonnanceur afin de poursuivre cette étude plus en profondeur.