JACQUETTE Pierrick
pierrick.jacquette@gmail.com

TP N°4 – Performances et Benchmarck

Temps :

Temps estimé à 8h et fini en 7h30 :

- 2h de recherche
- 3h pour l'implémentation
- 30min pour les graphiques
- 30min pour lmbench
- 1h30 pour le rapport

Exécution :

- make puis ./tp4 ou make run

Les options pour :

- afficher la liste des options disponibles : --help ou -h
- afficher les mesures de temps pour les processus : --processus ou -p
- afficher les mesures de temps pour les threads : --thread ou -t
- afficher le plus petit intervalle de temps mesurables : --delta ou -d
- afficher les mesures de temps d'allocation mémoire : --memoire ou -m
- spécifier le nombre d'opération par mesure avec X>99 : --nbMesure X ou -n X
- spécifier le nombre de mesure souhaité avec X>9 : --nbFois X ou -f X
- afficher les mesures de temps de création : --create ou -c
- afficher les mesures de temps de changement de context : --context ou -v
- ajouter les mesures de temps dans le fichier output.dat : --graph ou -g
- effectuer tout ce qui est possible sauf les graphiques : --all ou -a
- par défaut tout est effectué avec nbMesure vaut 700 et nbFois vaut 100.

- make install : pour pouvoir générer les graphiques

Environnement : (merci lscpu, dmidecode -t memory, /etc/lsb-release et gcc –version)

Environnement du second ordinateur :

Matériel :

- Processeur et sa vitesse: Intel(R) Core(TM) i7-6700K CPU at 4.00GHz
- Caches : L1 de 32KB, L2 de 256KB et L3 de 8192KB
- Mémoire et sa vitesse : 8192MB à 2133 MHz

Logiciel :

- Système : Ubuntu 14.04.5 LTS
- Compilateur : gcc 4.8.4

Quel outil pour quelle mesure ?

Avec plusieurs ramettes à ma disposition, j'en prends plusieurs je mesure la hauteur de toutes les ramettes (sans l'emballage:)). Je divise le résultat par le nombre de ramettes et par le nombre de feuilles dans une ramette pour avoir l'épaisseur moyenne d'une feuille (la plus précise possible).

Ce problème est bien évidemment en rapport avec la question posée dans ce TP. En effet, je dois mesurer des événements courts avec des outils pas forcément adéquates. Il faut donc effectuer plusieurs mesures.

Question optionnelle :

	Origine		Unité de temps	Précision	Point d'origine	Débordement
	Système	Bibliothèque				
time	✓		seconde	seconde	1er janvier 1970	Si la date courante est supérieure au type de données (en 2039)
gettimeofday	✓		seconde et microseconde	microseconde	1er janvier 1970	
clock_gettime		✓	seconde	nanoseconde		

J'ai calculé les deltas suivants :

- time : 1 sec
- gettimeofday : 0 sec - 1 usec
- gettime : 0 sec - 205 nsec

J'ai donc choisi d'implémenter mon programme avec la fonction clock_gettime comme timer en microseconde (us).

Remarques :

Quel type de temps aurais-je besoin pour ce TP ? Réponse : « Horloge murale ».

Avant de commencer l'implémentation j'ai essayé de comprendre tout ce que l'on me demandait, les subtilités, cela m'a permis de comprendre qu'il fallait une possibilité de générer des résultats dans un format comparable. Par ailleurs pour plusieurs parties, les mêmes fonctionnalités étaient demandés. J'ai fait des fonctions de temps définies à un endroit, un output en forme de tableau pour que l'utilisateur puissent les comparer et s'il le souhaite, générer des graphiques à partir de ces résultats (ce qui est beaucoup plus lisible). Les résultats sont écrits dans un fichier (avec une certaines syntaxe), pour que « gnuplot » puisse créer les graphiques correspondants.

Les temps observés sont faibles donc il faut un effectuer un certain nombre d'action pour un seul timer. Pour avoir un temps moyen et plus précis, j'ai décidé de faire n fois ce timer. Cela me permet en même temps de calculer un minimum et maximum. L'utilisateur peut modifier le nombre (nbMesure) où une action est exécutée et combien de fois il veut effectuer ce nombre (nbFois).

Lors de mes tests j'étais parfois en attente d'un calcul assez important surtout pour la création de processus, j'ai donc ajouté un pourcentage de progression pour chaque calcul.

Les perturbations possibles :

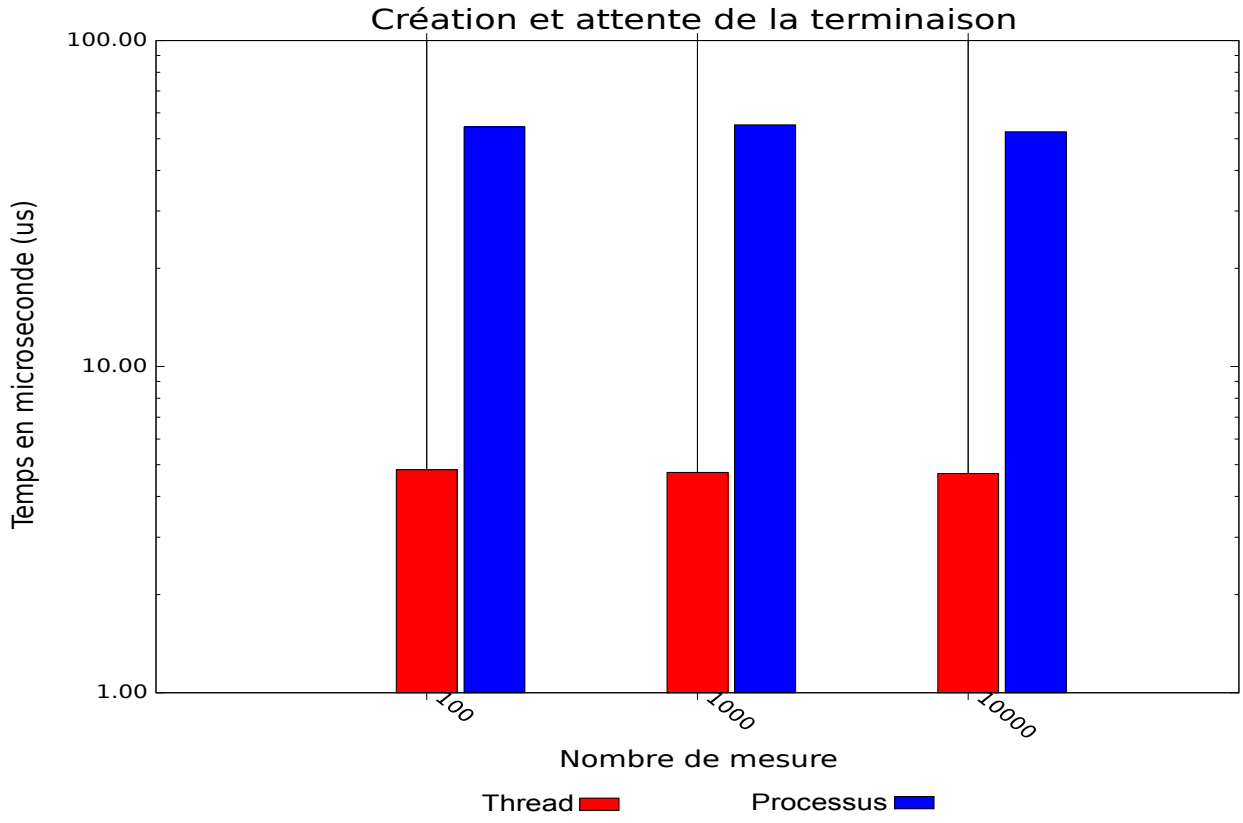
Les changements de processeur (donc les registres …) peuvent perturber la mesure du temps. Pour éviter cela je dédie un processeur à mon exécution. Cela me permet d'éviter les erreurs pouvant se produire avec des architectures matérielles modernes (multi-processeurs, multi-threading,..).

L'ordonnancement opéré par le système entre les différents processus exécutés sur la machine.

Création et attente de terminaison de processus et thread

Pour les threads le point de départ est la création d'un thread avant le pthread_create et le point final est après la fin de sa fonction. Soit avant la création du thread et dés que le thread est terminé. Pour les processus le point de départ est la création d'un processus avant le fork et le point final est après le waitpid pour le père. Soit avant la création du fils et dés que le fils est mort. Cela répond bien au sujet.

Création et attente de terminaison	Temps (en microseconde)								
	n = 100			n = 1 000			n = 10 000		
	Moyen	Min	Max	Moyen	Min	Max	Moyen	Min	Max
Thread	4.780	4.423	8.645	4.684	4.277	5.275	5.178	5.074	6.027
Processus	53.598	52.401	62.023	53.976	53.748	55.153	48.709	48.229	52.096



La différence de temps obtenue entre le temps de création et attente de terminaison d'un thread par rapport à un fork est due à la copie d'espace mémoire. Pour un thread_create l'espace d'adressage est partagé et les piles ne sont pas protégées. Pour le fork, le nouveau processus reçoit une copie des adresses du code, données, tas et pile.

Changement de contexte

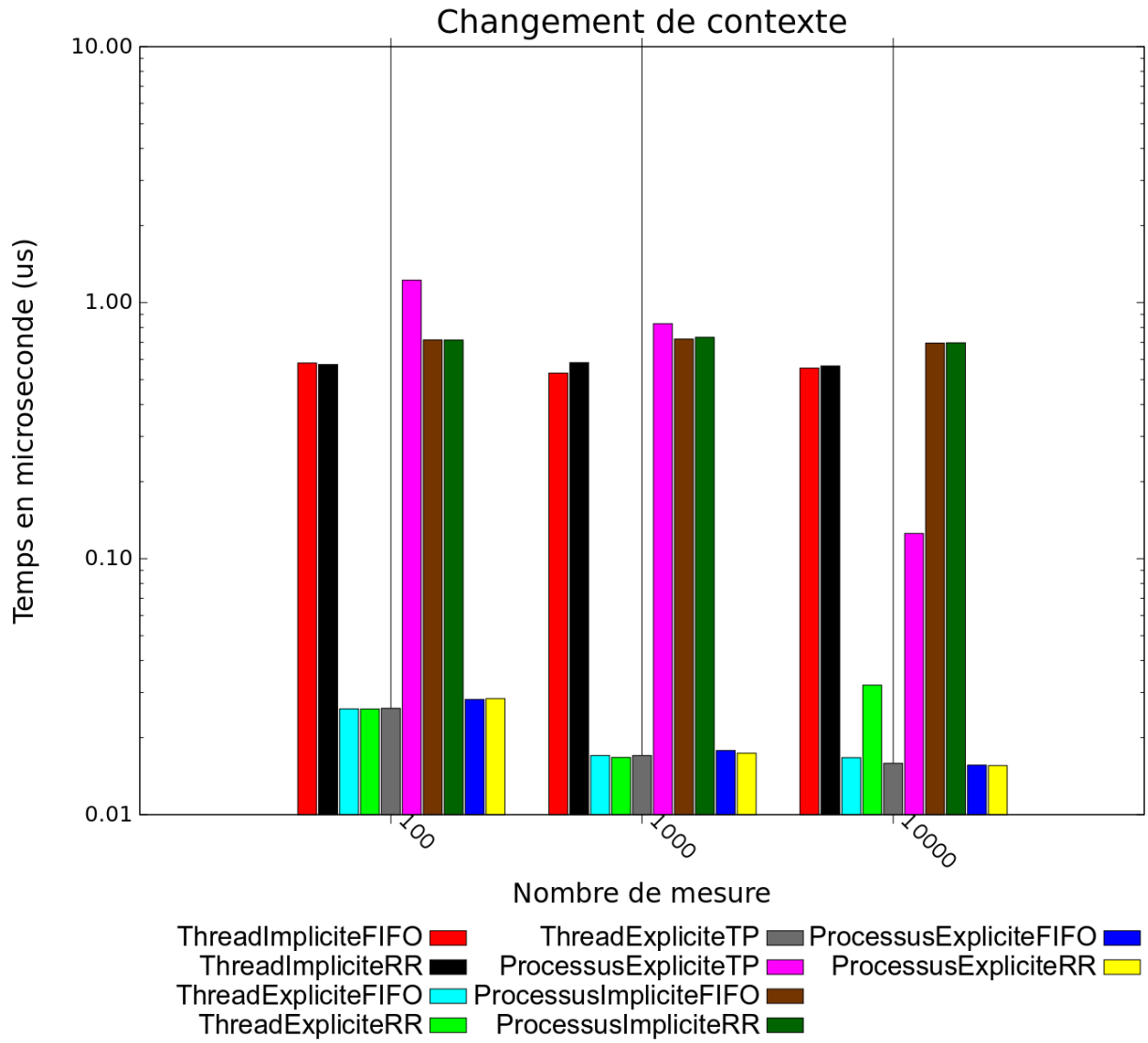
Afin d'obtenir l'ordonnancement désiré entre les processus ou threads, j'ai opté pour l'utilisation de sémaphores. Je calcule la priorité que je donne à chaque entité en prenant la valeur moyenne des possibilités.

Pour les threads le point de départ est la section critique d'un thread et le point final est la section critique de l'autre thread. Soit un changement de thread qui s'exécute. Pour les processus le point de départ est la section critique du fils et le point final est la section critique du père. Soit un changement de processus qui s'exécute. Cela répond bien au sujet. J'ai pris la liberté d'implémenter le changement de context round-robin.

Changement de contexte			Temps (en microseconde)								
			n = 100			n = 1 000			n = 10 000		
			Moyen	Min	Max	Moyen	Min	Max	Moyen	Min	Max
Thread	implicite	FIFO	0.591	0.562	0.746	0.522	0.463	0.647	0.546	0.532	0.599
		RR	0.585	0.579	0.619	0.589	0.570	0.625	0.566	0.558	0.578
	explicite	FIFO	0.027	0.009	1.729	0.017	0.001	1.651	0.015	0.001	1.589
		RR	0.027	0.009	1.716	0.017	0.001	1.644	0.015	0.001	1.589
		TP	0.027	0.009	1.717	0.016	0.001	1.601	0.016	0.001	1.654
Processus	implicite	FIFO	0.713	0.708	0.793	0.716	0.704	0.750	0.699	0.691	0.709
		RR	0.712	0.705	0.750	0.717	0.699	0.767	0.647	0.001	0.705
	explicite	FIFO	0.028	0.009	1.888	0.016	0.001	1.574	0.015	0.001	1.570
		RR	0.028	0.009	1.931	0.016	0.001	1.566	0.015	0.001	1.570
		TP	1.292	1.235	1.685	0.922	0.001	1.839	0.105	0.001	1.869

J'ai pu remarquer que les changements implicites pour les threads (sched_yield() avec PTHREAD_EXPLICIT_SCHED) ne fonctionnait pas, il exécute complètement un thread puis l'autre thread.

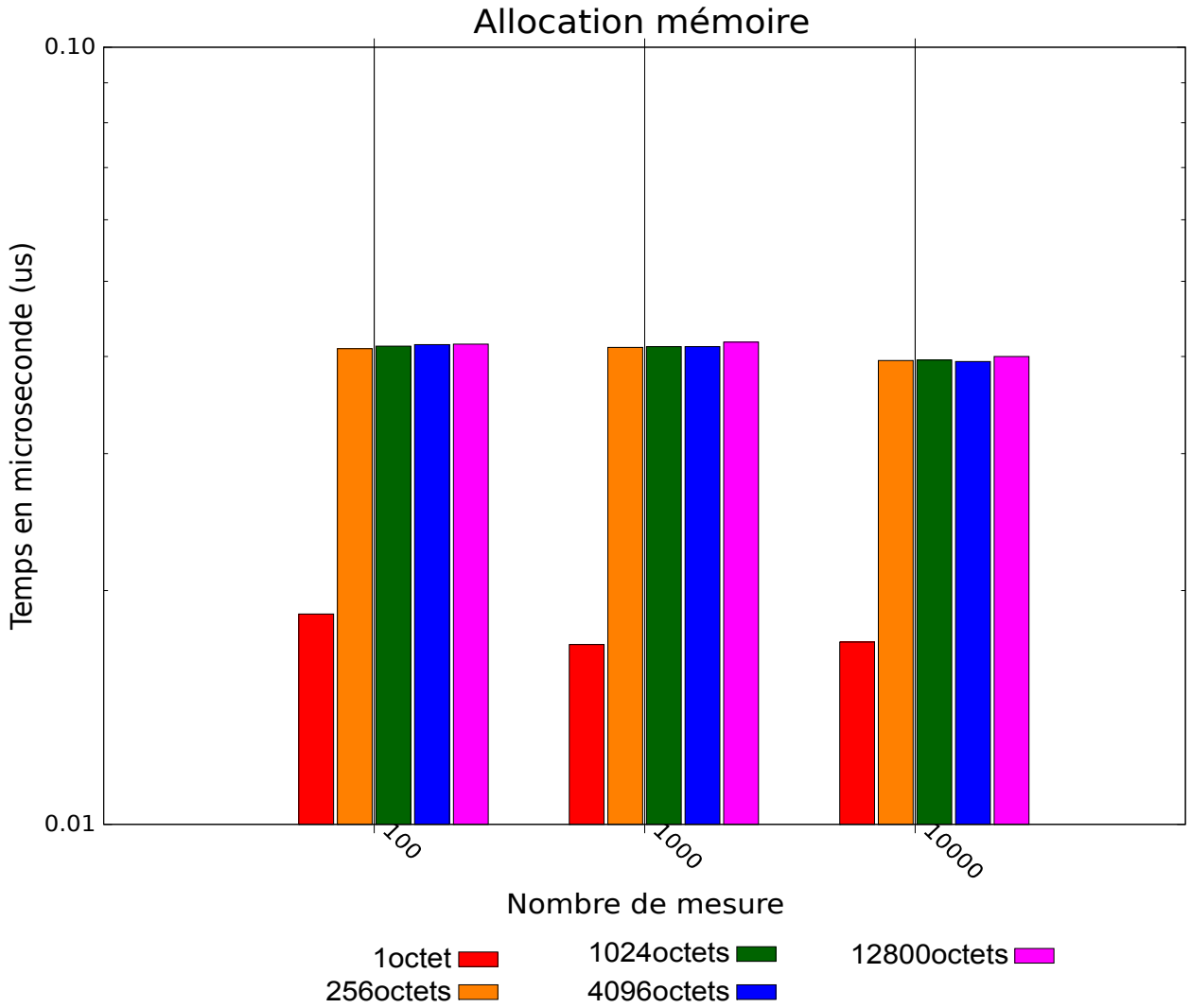
Je remarque que les changements de contexte des threads sont plus rapides que les changements de contexte pour que les processus. Les changements explicites en temps partagé sont plus long que ceux avec un ordonnancement FIFO et RR. Ils sont plus prioritaires qu'en temps partagé (cela permet d'éviter certaines interruptions).



Allocation mémoire

Pour la mémoire le point de départ est juste avant le malloc et le point de fin est juste après le malloc.

Mémoire (octets)	Temps (en microseconde)								
	n = 100			n = 1 000			n = 10 000		
	Moyen	Min	Max	Moyen	Min	Max	Moyen	Min	Max
1	0.018	0.017	0.026	0.017	0.016	0.032	0.017	0.015	0.020
256	0.041	0.041	0.046	0.041	0.039	0.054	0.039	0.039	0.043
1024	0.041	0.040	0.042	0.040	0.039	0.043	0.039	0.039	0.040
4096	0.041	0.040	0.049	0.040	0.039	0.041	0.039	0.039	0.043
128000	0.041	0.040	0.48	0.040	0.039	0.41	0.040	0.039	0.42



L'allocation mémoire se fait par page mémoire puis avec un offset. Peut-importe le nombre d'octets alloué, c'est un pointeur vers le début de l'espace mémoire qui est crée. Cette espace est choisi par les emplacements libre en mémoire. Mais tant que l'on accède pas à un octet de la mémoire, ce n'est que de la mémoire virtuelle.

Comparaison avec d'autres résultats de TP

J'ai testé mon programme sur deux ordinateurs différents, j'ai des résultats presque identiques. L'aspect général des graphiques est le même pour les deux ordinateurs. La différence entre les deux vient sûrement de la vitesse du CPU. Les graphiques des résultats obtenues sont mis dans le dossier (Graphique) pour économiser l'encre et le papier :)

Je pense que je n'ai pas encore toutes les compétences (et expériences) requises pour pouvoir garantir quoique ce soit vis-à-vis d'une quelconque garantie de ces indications de temps dans un système critique. Il faudrait utiliser un type de vérification avec model-checking et simulation.

Matériel :

- Processeur et sa vitesse: Intel(R) Core(TM) i5-6300HQ CPU at 2.30GHz
- Caches : L1 de 32KB, L2 de 256KB et L3 de 6144KB
- Mémoire et sa vitesse : 8192MB à 2133 MHz

Logiciel :

- Système : Ubuntu 16.04.2 LTS
- Compilateur : gcc 5.4.0

Outils de « benchmarking »

J'ai réussi à implémenter la création de thread pour lmbench. Le code est dans lmbench/lat_proc.c

Quoi		Temps (en microseconde)	
		Ordinateur à 2.30GHz	Ordinateur à 4.00GHz
Fork create	Mes résultats	69.0065	53.5982
	lmbench	90.6557	56.9176
Fork context	Mes résultats	0.80	0.92
	lmbench	0.59	0.83
Thread create	Mes résultats	7.0792	4.7805
	lmbench	8.3565	5.8630

Mes résultats de performances par rapport à ceux fournis par lmbench sont assez similaires (~ +/- 10%).