

Programmation Concurrente

Password Cracker

Travail Pratique

But du projet

Le but de ce travail pratique est de faire vos premiers pas de hacker, en utilisant la puissance des machines multi-processeurs afin de réduire le temps nécessaire pour cracker un mot de passe en attaque « brute force », c'est-à-dire en essayant toutes les combinaisons possibles.

On aimerait réaliser un programme multi-threadé, `crack`, prenant en argument un mot de passe crypté (aussi appelé *hash*), une chaîne de deux caractères *salt*, et enfin le nombre de threads que nous désirons utiliser.

Les deux caractères du *salt* sont utilisés pour créer un hash plus difficile à casser et ainsi augmenter la sécurité d'un système. Pour plus de détails sur la raison d'être du *salt*, voir :

[https://en.wikipedia.org/wiki/Salt_\(cryptography\)](https://en.wikipedia.org/wiki/Salt_(cryptography)).

Le fonctionnement du programme est simple : il s'agit de tester toutes les combinaisons de *hash* possibles afin de déterminer le mot de passe pour le *hash* passé en entrée. Pour ce faire, nous utilisons une fonction d'encryption qui encrypte le mot de passe à deviner. Cette fonction produit un *hash* et si celui-ci correspond au *hash* donné, alors le mot de passe a été deviné. L'approche par « brute force » nécessite donc de tester toutes les possibilités jusqu'à tomber sur le bon hash... ce qui est évidemment extrêmement inefficace, d'où les attaques par dictionnaires.

Cahier des charges

- Votre programme prendra en arguments le hash à cracker, le salt, ainsi que le nombre de threads à utiliser. A titre d'exemple, essayez de crackez le mot de passe encrypté par le hash « 431pugYzZc5QM » et le salt « 43 ».
- Votre programme indiquera le nombre de combinaisons totales ainsi que le nombre de threads utilisés.
- Votre programme affichera le temps total écoulé pour cracker le mot de passe.
- Essayez de rendre le travail de chaque thread le plus équitable possible.
- Seules les primitives de synchronisation vues en cours sont autorisées.
- **Evitez toute attente active.**
- Le programme se terminera une fois le mot de passe trouvé. A noter qu'aucun thread ne pourra utiliser la fonction `exit` pour terminer le programme.
- **Aucune** variable globale ne doit être utilisée.

Questions

1. Etablissez un graphe illustrant les performances de votre programme multi-threadé en variant le nombre de threads.
2. Que remarquez-vous ? Le gain de vitesse est-il linéaire avec le nombre de threads ?
3. Bonus : quel est l'impact de l'hyper-threading (coeurs logiques) sur les performances ?

Note pour la question 3 : la commande `lscpu` permet de lister le nombre de CPUs disponibles sur le système ainsi que le nombre de coeurs logiques par CPU. Dans l'exemple ci-dessous, le système possède un total de 4 CPUs logiques où chaque CPU physique possède 2 coeurs logiques :

```
Architecture:      x86_64
CPU op-mode(s):    32-bit, 64-bit
Byte Order:        Little Endian
CPU(s):            4
On-line CPU(s) list: 0-3
Thread(s) per core: 2
Core(s) per socket: 2
```

Informations utiles

Validité des caractères

Le domaine de validité des caractères du mot de passe sont les chiffres de '0' à '9', les caractères alphabétiques simples (sans accents) de 'a' à 'z' (minuscules et majuscules) ainsi que les caractères spéciaux '~', '!' et '*'. Vous pouvez vous aider de la table ASCII ci-dessous :

	30	40	50	60	70	80	90	100	110	120
0:	(2	<	F	P	Z	d	n	x	
1:)	3	=	G	Q	[e	o	y	
2:	*	4	>	H	R	\	f	p	z	
3:	!	+	5	?	I	S]	g	q	{
4:	"	,	6	@	J	T	^	h	r	
5:	#	-	7	A	K	U	_	i	s	}
6:	\$.	8	B	L	V	`	j	t	~
7:	%	/	9	C	M	W	a	k	u	DEL
8:	&	0	:	D	N	X	b	l	v	
9:	'	1	;	E	O	Y	c	m	w	

Encryption

La fonction `crypt_r` permet d'encrypter une chaîne de caractères avec un *salt* donné afin d'obtenir un *hash*, comme montré ci-dessous:

```
struct crypt_data *cdata = (struct crypt_data *)
malloc(sizeof(struct crypt_data));
cdata->initialized = 0;
char *hash = crypt_r(password, salt, cdata);
```

La fonction `crypt_r` nécessite l'inclusion du header `crypt.h` et de spécifier `#define _GNU_SOURCE` avant l'inclusion de `crypt.h`. Pour plus de détails, tapez « `man crypt_r` »

Important : le code ci-dessus requiert la librairie `crypt`. Pour ce faire, passez `-lcrypt` au compilateur `gcc` au moment où vous générez votre exécutable final.

Attention: n'utilisez pas la fonction `crypt`, car celle-ci n'est pas réentrante !

Timing

Les mesures de timing peuvent être effectuées avec la fonction `clock_gettime` de la librairie `<time.h>` comme montré ci-dessous :

```
struct timespec start, finish;
clock_gettime(CLOCK_MONOTONIC, &start);

... // code à mesurer

clock_gettime(CLOCK_MONOTONIC, &finish);
double elapsed = finish.tv_sec - start.tv_sec;
elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
```

Important : le code ci-dessus requiert la librairie `rt`. Pour ce faire, passez `-lrt` au compilateur `gcc` au moment où vous générez votre exécutable final.

Code

Le code à rendre doit respecter les consignes décrites dans le document « *Consignes pour l'écriture du code* » sur la page CyberLearn du cours.

Travail à rendre

N'oubliez pas de lire attentivement le document « *Consignes Travaux Pratiques.pdf* » disponible sur la page CyberLearn du cours.

Pour ce travail pratique, chaque groupe me rendra une archive contenant les fichiers suivants :

- Fichiers sources, incluant un `makefile` pour compiler le projet (et faire le ménage).
- Un graphe au format PDF illustrant les mesures de temps effectuées (cf. questions).
- Les réponses aux questions au format PDF (celles-ci peuvent être sur le même page que le graphe).