

## Atividade da Aula 05 (08/09/2021)

Aluno: Gabriel Hoffmann

Curso: Ciência da Computação

Refaça o exercício apresentado em aula, modificando para tornar as threads em loop e descreva os comportamentos considerando os comandos:

```
1  #include <pthread.h>
2  #include <stdio.h>
3  unsigned long soma[4];
4  void *thread_fn(void *arg)
5  {
6      long id = (long)arg;
7      int inicio = id * 2500000000;
8      int i = 0;
9      while (i < 2500000000)
10     {
11         soma[id] += (i + inicio);
12         i++;
13     }
14     return NULL;
15 }
16 int main(void)
17 {
18     pthread_t t1, t2, t3, t4;
19     unsigned long resultado = 0;
20     pthread_create(&t1, NULL, thread_fn, (void *)0);
21     pthread_create(&t2, NULL, thread_fn, (void *)1);
22     pthread_create(&t3, NULL, thread_fn, (void *)2);
23     pthread_create(&t4, NULL, thread_fn, (void *)3);
24     pthread_join(t1, NULL);
25     pthread_join(t2, NULL);
26     pthread_join(t3, NULL);
27     pthread_join(t4, NULL);
28     resultado = soma[0] + soma[1] + soma[2] + soma[3];
29     printf("%lu\n", resultado);
30     return 0;
31 }
```

Aumentei o laço de repetição para o programa entrar em loop e poder a partir de outro terminal verificar seu funcionamento com outros comandos

```
Ins gabrielsh2@DESKTOP-827LBKF:~/workspace$ gcc thread.c -o thread-05 -lpthread
M gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./thread-05
-bash: ./thread-05: No such file or directory
SS gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./thread-05
^C
Ins gabrielsh2@DESKTOP-827LBKF:~/workspace$ gcc thread.c -o thread-loop -lpthread
★ gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./thread-loop
```

gabrielsh2@DESKTOP-827LBKF: ~

```
top - 18:34:58 up 22 min, 0 users, load average: 0.52, 0.58, 0.59
Tasks: 7 total, 1 running, 6 sleeping, 0 stopped, 0 zombie
%Cpu(s): 60.0 us, 0.5 sy, 0.0 ni, 39.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 16326.3 total, 9650.0 free, 6452.3 used, 224.0 buff/cache
MiB Swap: 49152.0 total, 49042.7 free, 109.3 used. 9743.4 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
232	gabriel+	20	0	43472	628	456	S	353.3	0.0	0:10.60	thread-loop
1	root	20	0	9784	840	480	S	0.0	0.0	0:03.12	init
7	root	20	0	8940	220	180	S	0.0	0.0	0:00.00	init
8	gabriel+	20	0	18076	3616	3512	S	0.0	0.0	0:00.10	bash
201	root	20	0	9784	236	188	S	0.0	0.0	0:00.01	init
202	gabriel+	20	0	18076	3588	3476	S	0.0	0.0	0:00.04	bash
215	gabriel+	20	0	18920	2148	1528	R	0.0	0.0	0:00.08	top

Utilizando o top podemos observar o programa rodando em loop com o PID 232, além de outros dados disponibilizados pelo top como o tempo de execução.

```
gabrielsh2@DESKTOP-827LBKF:~$ ps -eLf
```

UID	PID	PPID	LWP	C	NLWP	STIME	TTY	TIME	CMD
root	1	0	1	0	7	18:12	?	00:00:00	/init
root	1	0	6	0	7	18:12	?	00:00:00	/init
root	1	0	78	0	7	18:19	?	00:00:00	/init
root	1	0	79	0	7	18:20	?	00:00:00	/init
root	1	0	81	0	7	18:20	?	00:00:00	/init
root	1	0	83	0	7	18:20	?	00:00:00	/init
root	1	0	174	0	7	18:27	?	00:00:00	/init
root	7	1	7	0	1	18:12	tty1	00:00:00	/init
gabriel+	8	7	8	0	1	18:12	tty1	00:00:00	-bash
root	201	1	201	0	1	18:30	tty2	00:00:00	/init
gabriel+	202	201	202	0	1	18:30	tty2	00:00:00	-bash
gabriel+	232	8	232	0	5	18:34	tty1	00:00:00	./thread-loop
gabriel+	232	8	233	99	5	18:34	tty1	00:02:16	./thread-loop
gabriel+	232	8	234	99	5	18:34	tty1	00:02:16	./thread-loop
gabriel+	232	8	235	99	5	18:34	tty1	00:02:16	./thread-loop
gabriel+	232	8	236	99	5	18:34	tty1	00:02:16	./thread-loop
gabriel+	237	202	237	0	1	18:37	tty2	00:00:00	ps -eLf

```
gabrielsh2@DESKTOP-827LBKF:~$
```

Com o comando ps -eLf podemos ver as threads separadas rodando no mesmo tempo de 2 minutos e 16 segundos, além de outros dados sobre o processo.

Entrando no /proc do processo na pasta do seu PID temos acesso a vários outros dados

```
gabrielsh2@DESKTOP-827LBKF:/proc/232$ ls
attr      cmdline  environ  gid_map  mountinfo net       oom_score_adj  setgroups  statm  uid_map
auxv      comm     exe      limits   mounts   ns        root           smaps      status
cgroup    cwd      fd       maps     mountstats oom_adj    schedstat     stat       task
```

Um exemplo dos diversos dados que podem ser lidos, usando o vim no status temos informações detalhadas do processo onde podemos ver seu número de threads ou o limits em que temos informações sobre as limitações do quanto a máquina disponibilizou para o processo.

```
Name:      thread-loop
State:     S (sleeping)
Tgid:      232
Pid:       232
PPid:      8
TracerPid: 0
Uid:       1000    1000    1000    1000
Gid:       1000    1000    1000    1000
FDSize:    3
Groups:
VmPeak:    0 kB
VmSize:    43472 kB
VmLck:     0 kB
VmHWM:     0 kB
VmRSS:     628 kB
VmData:    0 kB
VmStk:     0 kB
VmExe:     4 kB
VmLib:     0 kB
VmPTE:     0 kB
Threads:   5
SigQ:      0/0
SigPnd:    0000000000000000
ShdPnd:    0000000000000000
SigBlk:    0000000000000000
SigIgn:    0000000000000000
SigCgt:    0000000000000000
"status" [readonly] 37L, 660C
```

Limit	Soft Limit	Hard Limit	Units
Max cpu time	unlimited	unlimited	seconds
Max file size	unlimited	unlimited	bytes
Max data size	unlimited	unlimited	bytes
Max stack size	8388608	unlimited	bytes
Max core file size	0	unlimited	bytes
Max resident set	unlimited	unlimited	bytes
Max processes	8041	8041	processes
Max open files	1024	4096	files
Max locked memory	65536	65536	bytes
Max address space	unlimited	unlimited	bytes
Max file locks	unlimited	unlimited	locks
Max pending signals	8041	8041	signals
Max msgqueue size	819200	819200	bytes
Max nice priority	40	40	
Max realtime priority	0	0	
Max realtime timeout	unlimited	unlimited	us

## Exercícios teóricos

A correta utilização de processos e *threads* é fundamental para garantir o desempenho e a transparência de sistemas distribuídos. Sobre esse tema, considere as afirmativas a seguir.

I. A sobreposição de *threads* em um processo é o principal recurso para obtenção de alto grau de transparência de distribuição em redes com longos tempos de propagação de mensagens.

II. A desvantagem de se estruturar um programa para utilizar múltiplas *threads* é que ele ficará dependente de sistemas multiprocessadores.

III. O modelo de *threads* implementado pelo sistema operacional deve ser aquele em que o gerenciamento de *threads* fica inteiramente no espaço de cada processo para evitar trocas de contexto entre processos e o núcleo (*kernel*) no chaveamento de *threads*.

IV. Servidores *multithreaded* têm melhor desempenho se estruturados com ao menos uma *thread* despachante e várias *threads* operárias para recebimento e processamento de requisições.

Assinale a alternativa correta.

### b) Apenas I e IV estão corretas

Considere o seguinte programa com dois processos concorrentes. O escalonador poderá alternar entre um e outro, isto é, eles poderão ser intercalados durante sua execução. As variáveis x e y são compartilhadas pelos dois processos e inicializadas antes de sua execução.

```
programa P
int x = 0;
int y = 0;
processo A {
    while (x == 0);
    print('a');
    y = 1;
    y = 0;
    print('d');
    y = 1;
}

processo B {
    print('b');
    x = 1;
    while (y == 0);
    print("c");
}
```

As possíveis saídas são:

### (b) badc ou bacd