

Vláknové programování

část III

Lukáš Hejmánek, Petr Holub
{`xhejtman, hopet`}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192
2013-04-08

Přehled přednášky

Další nástroje pro synchronizaci

Afinita

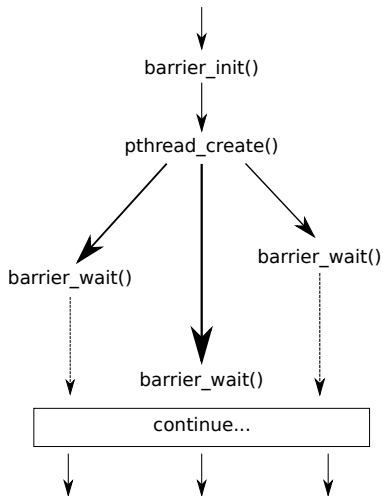
Atributy funkcí pthread knihovny

Další nástroje pro synchronizaci

Bariéry

- Bariéry jsou v podstatě místa setkání.
- Bariéra je místo, kde se vlákna setkají.
- Bariéra zablokuje vlákno do doby než k bariéře dorazí všechna vlákna.
- Příklad:
 - Vlákňové násobení matic: $M \times N \times O \times P$
 - Každé vlákno násobí a sčítá příslušný sloupec a řádek.
 - Po vynásobení $M \times N$ se vlákna setkají u *bariéry*.
 - Vynásobí předchozí výsledek $\times O$, opět se setkají u bariéry.
 - Dokončí výpočet vynásobením výsledku $\times P$.
- Ne všechny implementace POSIX threads poskytují bariéry!

Bariéry



Bariéry

- Datový typ `pthread_barrier_t`.
- Inicializace `pthread_barrier_init()`
(Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Při inicializaci specifikujeme, pro kolik vláken bude bariéra sloužit.
- Zastavení na bariéře `pthread_barrier_wait()`.
- Zrušení bariéry `pthread_barrier_destroy`.

Příklad bariéry

```
1 #include <pthread.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 pthread_barrier_t barrier;
6
7 void *
8 foo(void *arg) {
9     int slp = (int)arg;
10    printf("Working..\n");
11    sleep(slp);
12    printf("Waiting on barrier\n");
13    pthread_barrier_wait(&barrier);
14    printf("Synchronized\n");
15    return NULL;
16 }
```

Příklad bariéry

```
17 int
18 main(void)
19 {
20     pthread_t t1, t2;
21
22     pthread_barrier_init(&barrier, NULL, 2);
23     pthread_create(&t1, NULL, foo, (void*)2);
24     pthread_create(&t2, NULL, foo, (void*)4);
25
26     pthread_join(t1, NULL);
27     pthread_join(t2, NULL);
28     pthread_barrier_destroy(&barrier);
29     return 0;
30 }
```


Read Write zámky

- Read Write zámky dovolují násobné čtení ale jediný zápis.
- Příklad:
 - Několik vláken čte nějakou strukturu (velmi často!).
 - Jedno vlákno ji může měnit (velmi zřídka!).
 - Pozorování:
 - Je zbytečné strukturu zamykat mezi čtecími vlákny
Nemohou ji měnit a netvoří tedy kritickou sekci.
 - Je nutné strukturu zamknout, mění-li ji zapisovací vlákno
V této chvíli nesmí strukturu ani nikdo číst (není změněna atomicky).

Read Write zámky

- Nastupují Read Write zámky.
- Pravidla:
 - Není-li zámeček zamčen v režimu *Write*, může být libovolněkrát zamčen v režimu *Read*.
 - Je-li zámeček zamčen v režimu *Write*, nelze jej už zamknout v žádném režimu.
 - Je-li zámeček zamčen v režimu *Read*, nelze jej zamknout v režimu *Write*.
- Opět ne všechny implementace POSIX threads implementují RW zámky (korektně)!

RW zámky

- Datový typ **pthread_rwlock_t**.
- Inicializace **pthread_rwlock_init()**
(Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Zamknutí v režimu *Read* **pthread_rwlock_rdlock()**.
- Zamknutí v režimu *Write* **pthread_rwlock_wrlock()**.
- Opakované zamčení jednoho zámku stejným vláknem skončí chybou **EDEADLK**.
Není možné povýšit *Read* zámeček na *Write* zámeček a naopak.
- Odemknutí v libovolném režimu **pthread_rwlock_unlock()**
Pthreads nerozlišují odemknutí dle režimů, některé implementace vláken párují rdlock s příslušným rdunlock, stejně tak pro wrlock.
- Zrušení rw zámku **pthread_rwlock_destroy**.

Příklad

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4
5 struct x_t {
6     int a;
7     int b;
8     pthread_rwlock_t lock;
9 };
10
11 struct x_t x;
12
13 int quit = 0;
14
15 pthread_barrier_t start;
```

Příklad

```
16 void *
17 reader(void *arg)
18 {
19     int n = (int)arg;
20     pthread_barrier_wait(&start);
21
22     while(!quit) {
23         pthread_rwlock_rdlock(&x.lock);
24         if((x.a + x.b)%n == 0)
25             printf(".");
26         else
27             printf("+");
28         pthread_rwlock_unlock(&x.lock);
29         fflush(stdout);
30         sleep(1);
31     }
32     return NULL;
33 }
34 }
```

Příklad

```
35
36 void *
37 writer(void *arg)
38 {
39     int i;
40     pthread_barrier_wait(&start);
41     for(i=0; i < 10; i++) {
42         pthread_rwlock_wrlock(&x.lock);
43         x.a = i;
44         x.b = (i % 2)+1;
45         pthread_rwlock_unlock(&x.lock);
46         sleep(5);
47     }
48     quit = 1;
49     return NULL;
50 }
```

Příklad

```
52
53 int
54 main(void)
55 {
56     pthread_t t1, t2, t3;
57
58     x.a = 1;
59     x.b = 2;
60     pthread_rwlock_init(&x.lock, 0);
61     pthread_barrier_init(&start, NULL, 3);
62     pthread_create(&t1, NULL, reader, (void*)2);
63     pthread_create(&t2, NULL, reader, (void*)3);
64     pthread_create(&t3, NULL, writer, NULL);
65     pthread_join(t1, NULL);
66     pthread_join(t2, NULL);
67     pthread_join(t3, NULL);
68     pthread_rwlock_destroy(&x.lock);
69     pthread_barrier_destroy(&start);
70     return 0;
71 }
```

Problémy RW zámků

- Nebezpečí stárnutí zámků.
- Pokud je zamčená část kódu vykonávána déle než nezamčená, nemusí se nikdy podařit získat některý ze zámků.
- V předchozím příkladě nesmí být **sleep()** v zamčené části kódu!

```
1     for(i=0; i < 10; i++) {
2         pthread_rwlock_wrlock(&x.lock);
3         x.a = i;
4         x.b = (i % 2)+1;
5         pthread_rwlock_unlock(&x.lock);
6         sleep(5);
7     }
```


Try varianty synchronizace

- Pomocí try variant volání lze zjistit, zda vstup do kritické sekce je volný či nikoli.
- Funkce atomicky zkusí provést synchronizaci (např. zamknout zámek).
- V případě neúspěchu není funkce blokuující, ale okamžitě provede návrat.
- Neúspěch je signalizován návratovým kódem funkce (dle manuálové stránky, pro jednotlivá volání se může *lišit!*).
- Try varianty:
 - `pthread_mutex_trylock()`
 - `pthread_spin_trylock()`
 - `pthread_rwlock_tryrdlock()`
 - `pthread_rwlock_trywrlock()`
 - `sem_trywait()`

Příklad try zámku

```
1 #include <errno.h>
2 #include <stdio.h>
3 #include <pthread.h>
4
5 pthread_mutex_t lock;
6
7 void
8 foo(void)
9 {
10     if(pthread_mutex_trylock(&lock) == EBUSY) {
11         printf("Cannot acquire the lock right now\n");
12     } else {
13         printf("Locked\n");
14     }
15 }
16
17 int
18 main()
19 {
20     pthread_mutex_init(&lock, NULL);
21     foo();
22     foo();
23 }
```

Timed varianty synchronizace

- Nástroje zabraňující „věčnému“ čekání.
- Příklad:
 - Jak ukončit vlákno čekající na zámek pomocí globální proměnné?
- K většině blokujících synchronizačních rozhraní existují ekvivalentní rozhraní s časovým omezením.
- Po vypršení časového omezení je vrácena chyba návratovým kódem (dle manuálové stránky, pro jednotlivá volání se může *lišit!*).
- Timed varianty:
 - `pthread_cond_timedwait()`
 - `pthread_mutex_timedlock()`
 - `pthread_rwlock_timedrdlock()`
 - `pthread_rwlock_timedwrlock()`
 - `sem_timedwait()`
- Ne všechny implementace poskytují *timed* varianty.

Příklad timed zámku

```
1 #include <time.h>
2 #include <errno.h>
3 #include <stdio.h>
4 #include <pthread.h>
5
6 pthread_mutex_t lock;
7
8 void
9 foo(void)
10 {
11     struct timespec tsp;
12
13     tsp.tv_sec = time(NULL)+5;
14     tsp.tv_nsec = 0;
15     if(pthread_mutex_timedlock(&lock, &tsp) == ETIMEDOUT) {
16         printf("Timeout expired\n");
17     } else {
18         printf("Locked\n");
19     }
20 }
21
22 int
23 main()
24 {
25     pthread_mutex_init(&lock, NULL);
26     foo();
27     foo();
28 }
```

Jednorázové zavolání funkce

- Funkce typu inicializace chceme zavolat jen jednou.
- Konstrukce s příznakem, zda inicializace ještě nebyla provedena a následná inicializace je race condition.
- Zamykaní zbytečně snižuje paralelismus.
- **pthread_once()** zavolá jednou danou funkci.

Příklad jednorázového zavolání

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4
5 pthread_once_t once_init = PTHREAD_ONCE_INIT;
6 char initialized=0;
7
8 void
9 init(void)
10 {
11     /* do initialization */
12     printf("Initialized\n");
13     initialized = 1;
14 }
15
16 int
17 main(void)
18 {
19     if(!initialized) {
20         pthread_once(&once_init, init);
21     }
22     return 0;
23 }
```

Afinita

Architektura systému

- Vývoj
 - Single procesor
 - SMP systémy (symetrický multiprocessing – více rovnocenných procesorů)
 - Obvykle společná paměť
 - Všechny procesory mají do paměti „stejně daleko“
 - NUMA systémy (více procesorů, nejsou rovnocenné)
 - Procesory mají lokální paměť
 - Přístup do ne-lokální paměti přes ostatní CPU
 - SMT systémy (symetrický multithreading – procesory mají více jader)

Architektura systému

- Procesory mají lokální cache
- SMT systémy mívají lokální cache pro jádro a společnou cache pro více jader

Nastavení afinity

- Vlákno může být obecně plánováno na libovolný procesor
- Nemusí být vždy vhodné
- Migrace mezi procesory bývá poměrně drahá
- Pokud jde o výkon aplikace, můžeme chtít zabránit migraci procesů/vláken
- Úskalí ve statickém přiřazení procesů/vláken na procesor
- Afinita – nastavení množiny procesorů, na kterých má proces/vlákno běžet

Nastavení afinity

- Nastavení afinity procesů
 - `sched_setaffinity()` ;
 - `sched_getaffinity()` ;
 - Nelze použít pro vlákna
- Nastavení afinity vláknům
 - `pthread_setaffinity_np()` ;
 - `pthread_getaffinity_np()` ;
- Svázání procesů/vláken a CPU je realizováno pomocí CPU SET

CPU SET

- Množina typu `cpu_set_t` do níž přidáváme/odebíráme CPU
- CPU číslováme od 0
- `cpu_set_t *CPU_ALLOC();`
- `void CPU_ZERO_S();`
- `void CPU_SET_S();`
- `void CPU_CLR_S();`
- `int CPU_ISSET_S();`
- `void CPU_COUNT_S();`
- Logické operace mezi dvěma `cpu_set_t`: AND, OR, XOR, EQUAL

Příklad

```
1 #define _GNU_SOURCE
2 /* Musi byt jako prvni pred vsemi ostatnimi include */
3 #include <pthread.h>
4 #include <sched.h>
5 #define NUM_CPU 8
6
7 int
8 main()
9 {
10     cpu_set_t * set;
11
12     set = CPU_ALLOC(NUM_CPU);
13
14     CPU_ZERO_S(NUM_CPU, set);
15
16     CPU_SET(0, set);
17
18     pthread_setaffinity_np(pthread_self(), NUM_CPU, set);
19
20     CPU_FREE(set);
21
22     return 0;
23 }
```

Afinita před spuštěním aplikace

- Nastavení afinity u hotové aplikace
- `numactl (8), taskset (1)`
- Příklad
 - `numactl -cpubind=0 aplikace`
`taskset 0x1 aplikace`
 - Pustí aplikaci výhradně na CPU 0

Atributy funkcí pthread knihovny

Start vlákna

- **pthread_create()** funkci můžeme předávat atributy pro nově vytvářené vlákno.
- Atributy ovlivňují tři základní oblasti:
 - Osamostatnění vlákna
 - Nastavování priorit plánovače
 - Nastavení zásobníku
- Datový typ atributu **pthread_attr_t**.
- Inicializace **pthread_attr_init()**.
- Zrušení **pthread_attr_destroy()**.

Start vláknů – osamostatnění

- Osamostatněné vlákno uvolní všechny své zdroje jakmile skončí.
- Neosamostatněné vlákno je uvolní až při zavolání **pthread_join()**.
- Implicitně je každé vlákno neosamostatněné.
- **pthread_attr_setdetachstate()** nastaví vlákno osamostatněné (**PTHREAD_CREATE_DETACHED**) nebo neosamostatněné (**PTHREAD_CREATE_JOINABLE**).

Příklad osamostatnění

```
1 #include <pthread.h>
2
3 void *
4 foo(void * arg)
5 {
6     return NULL;
7 }
8
9 int
10 main(void)
11 {
12     pthread_t t;
13     pthread_attr_t attr;
14
15     pthread_attr_init(&attr);
16     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
17
18     pthread_create(&t, &attr, foo, NULL);
19     pthread_attr_destroy(&attr);
20     return 0;
21 }
```

Start vlákna – nastavení priority plánovače

- Pro vlákna lze do jisté míry ovlivnit způsob plánování.
- Lze nastavit tři základní pravidla plánování:
 - **SCHED_OTHER** – vlákno je plánováno dle standardního jaderného plánovače.
 - **SCHED_FIFO** – vlákno je plánováno dokud samo neskončí, nezablokuje se nebo není zrušeno.
 - **SCHED_RR** – vlákno je plánováno dokud samo neskončí, nezablokuje se, není zrušeno nebo nevyprší přidělené časové kvantum.

Start vlákna – nastavení priority plánovače

- Pro pravidla lze dále nastavit prioritu.
- Abychom mohli prioritu plánování olivnit, je třeba nastavit explicitní plánování na **PTHREAD_EXPLICIT_SCHED**:
 - **pthread_attr_setinheritsched()**
- Nastavení priority blízké *realtime* prioritě (**SCHED_FIFO**, **SCHED_RR**) může udělat pouze proces s právy administrátora.
- Realtime procesy mají přednost před ostatními.

Příklad nastavení priorit plánovače

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <string.h>
5
6 volatile int quit = 0;
7
8 void *
9 foo(void *arg)
10 {
11     long i=0;
12
13     while(!quit) {
14         i++;
15         if((i % 10000) == 0)
16             usleep(10);
17     }
18     return (void*)i;
19 }
```

Příklad nastavení priorit plánovače

```
19 int
20 main(void)
21 {
22     pthread_t t[3];
23     pthread_attr_t attr;
24     struct sched_param param;
25     long res;
26     int i, prio=10;
27
28     memset(&param, 0, sizeof(param));
29
30     pthread_attr_init(&attr);
31
32     pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
33     pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
34
35     for(i=0; i < 3; i++) {
36         param.sched_priority = prio;
37         pthread_attr_setschedparam(&attr, &param);
38         pthread_create(&t[i], &attr, foo, NULL);
39         prio+=10;
40     }
```

Příklad nastavení priorit plánovače

```
46
47     sleep(4);
48     quit = 1;
49     prio=10;
50     for(i=0; i < 3; i++) {
51         pthread_join(t[i], (void*)&res);
52         printf("Thread_with_prio_%d_%ld_iterations\n", prio, res);
53         prio+=10;
54     }
55     return 0;
56 }
```

Výstup příkladu

- Thread with prio 10 39930000 iterations
- Thread with prio 20 65870000 iterations
- Thread with prio 30 103812132 iterations
- Poznámka:
 - Vynechání volání **usleep()** má za následek takřka zablokování systému.
 - Z tohoto důvodu je povoleno nastavit realtime priority pouze administrátorským procesům.
 - Při jejich programování je nutno brát ohled na preempci ostatních procesů.

Nastavení zásobníku

- Implicitní velikost zásobníku pro vlákno je v Linuxu 8 MB.
- Chceme-li vytvořit 1000 vláken, potřebovali bychom 8 GB paměti jen pro zásobníky vláken.
- Pthread knihovna umožňuje změnit velikost zásobníku pro vlákno.
- **pthread_attr_setstacksize()**.
- Je nutné nastavit velikost zásobníku tak, aby se na něj vešly lokální proměnné všech funkcí, které se po sobě mohou zavolat. V opačném případě obdržíme signál **SIGSEGV** při vstupu do funkce, jejíž proměnné se na zásobník už nevlézou. Chyba vypadá na první pohled dost záhadně!

Příklad nastavení zásobníku

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4 void*
5 foo(void* arg)
6 {
7     return NULL;
8 }
9
10 int
11 main(void)
12 {
13     pthread_attr_t attr;
14     pthread_t t;
15
16     pthread_attr_init(&attr);
17
18     pthread_attr_setstacksize(&attr, 65536);
19
20     pthread_create(&t, &attr, foo, NULL);
21
22     pthread_join(t, NULL);
23     pthread_attr_destroy(&attr);
24
25     return 0;
26 }
```

Atributy mutexů – inverze priorit

- Mějme 2 vlákna s různou prioritou, A nechtť má vysokou prioritu, B nechtť má nízkou prioritu.
- Nechtť B zamkne sdílený objekt. Pak A při přístupu ke sdílenému objektu je zdržováno nízkou prioritou vlákna B i přes vlastní vysokou prioritu.

Atributy mutexů

- Prioritní protokoly zámeků.
- Lze specifikovat:
 - Žádný protokol (implicitní chování) **PTHREAD_PRIO_NONE**.
 - Protokol dědění priorit **PTHREAD_PRIO_INHERIT**.
 - Vlákno po zamčení objektu získá (zdědí) prioritu od vlákna s nejvyšší prioritou.
 - Protokol omezení priorit **PTHREAD_PRIO_PROTECT**.
 - Vlákno po zamčení objektu získá definovanou prioritu (nastavenou pomocí **pthread_mutex_setprioceiling**), je-li tato vyšší než jeho aktuální.
 - Změna pomocí volání **pthread_mutexattr_setprotocol()**.
 - Po odemčení objektu vlákno získá svou původní prioritu.
- Mechanismus protokolů a priorit je silně implementačně závislý, není všude podporován!

Atributy mutexů

- Datový typ `pthread_mutexattr_t`.
- Inicializace `pthread_mutexattr_init()`.
- Zrušení atributu `pthread_mutexattr_destroy()`.

Příklad prioritních mutexů

- Nutno kompilovat: `gcc -g -o priomutex priomutex.c -pthread -D__REENTRANT -D_XOPEN_SOURCE=500`

```
1 #include <pthread.h>
2
3 int
4 main()
5 {
6     pthread_mutex_t lock;
7     pthread_mutexattr_t attr;
8
9     pthread_mutexattr_init(&attr);
10    pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_PROTECT);
11    pthread_mutexattr_setprioceiling(&attr, 10);
12
13    pthread_mutex_init(&lock, &attr);
14
15    pthread_mutex_destroy(&lock);
16    pthread_mutexattr_destroy(&attr);
17    return 0;
18 }
```

Atributy mutexů

- Definování chování zámků v případě násobného zamčení/odemčení stejným vláknem.
- Ve většině případů je pokus zamknout mutex vláknem, které již tento zámek drží, chybou končící deadlockem.
- Pthreads umožní nastavit chování v takových případech.

Atributy mutexů

- **PTHREAD_MUTEX_NORMAL** vlákno se při násobném zamčení zámku deadlockne. Výsledek odemčení nezamčeného zámku není definován.
- **PTHREAD_MUTEX_ERRORCHECK** vláknu je vrácena chyba při pokusu o násobné zamčení zámku. *Je nutno kontrolovat návratový kód funkce zamčení zámku!* Pokus o odemčení nezamčeného zámku je signalizován chybovým návratovým kódem.
- **PTHREAD_MUTEX_RECURSIVE** vlákno smí provést násobné zamčení zámku. Výsledek je stejný jako by byl zámek zamčen jen jednou. Při pokusu o odemčení nezamčeného zámku je vrácena chyba.
- **PTHREAD_MUTEX_DEFAULT** násobné zamčení i odemčení nemají definované chování. Tato volba je implicitní.

Příklad násobných zámků

- Nutno kompilovat: `gcc -g -o multilock multilock.c -pthread -D__REENTRANT -D_XOPEN_SOURCE=500`

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 int
5 main()
6 {
7     pthread_mutexattr_t attr;
8     pthread_mutex_t lock;
9
10    pthread_mutexattr_init(&attr);
11    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
12
13    pthread_mutex_init(&lock, &attr);
14
15    printf("Return_code_of_the_first_lock_%d\n",
16          pthread_mutex_lock(&lock));
17    printf("Return_code_of_the_second_lock_%d\n",
18          pthread_mutex_lock(&lock));
19
20    pthread_mutex_unlock(&lock);
21    pthread_mutex_destroy(&lock);
```

Příklad násobných zámků

```
21
22     pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK);
23
24     pthread_mutex_init(&lock, &attr);
25
26     printf("Return_code_of_the_first_lock_%d\n",
27           pthread_mutex_lock(&lock));
28     printf("Return_code_of_the_second_lock_%d\n",
29           pthread_mutex_lock(&lock));
30
31     pthread_mutex_unlock(&lock);
32     pthread_mutex_destroy(&lock);
33
34     pthread_mutexattr_destroy(&attr);
35     return 0;
36 }
```

Výstup programu

- Rekurzivní zámek:
Return code of the first lock 0
Return code of the second lock 0
- Errorcheck:
Return code of the first lock 0
Return code of the second lock 35

Atributy podmínek

- Atribut podmínek umožňuje specifikovat typ času pro volání **`pthread_cond_timedwait()`**.
- Implicitně se timeout podmínky řídí dle hodin reálného času.
- Hodiny reálného času mohou skákat dopředu ale i dozadu, což může způsobit problémy.
- Proto lze pro podmínky specifikovat časový zdroj **`CLOCK_MONOTONIC`**, který se vždy pouze zvyšuje.

Atributy podmínek

- Datový typ `pthread_condattr_t`.
- Inicializace `pthread_condattr_init()`.
- Nastavení/zjištění zdroje času
`pthread_condattr_setclock()`,
`pthread_condattr_getclock()`.
- Zrušení atributu `pthread_condattr_destroy()`.

Příklad atributů podmínek

```
1 #include <time.h>
2 #include <pthread.h>
3
4 int
5 main()
6 {
7     pthread_condattr_t attr;
8     pthread_cond_t cond;
9
10    pthread_condattr_init(&attr);
11
12    pthread_condattr_setclock(&attr, CLOCK_MONOTONIC);
13
14    pthread_cond_init(&cond, &attr);
15
16    pthread_cond_destroy(&cond);
17    pthread_condattr_destroy(&attr);
18    return 0;
19 }
```