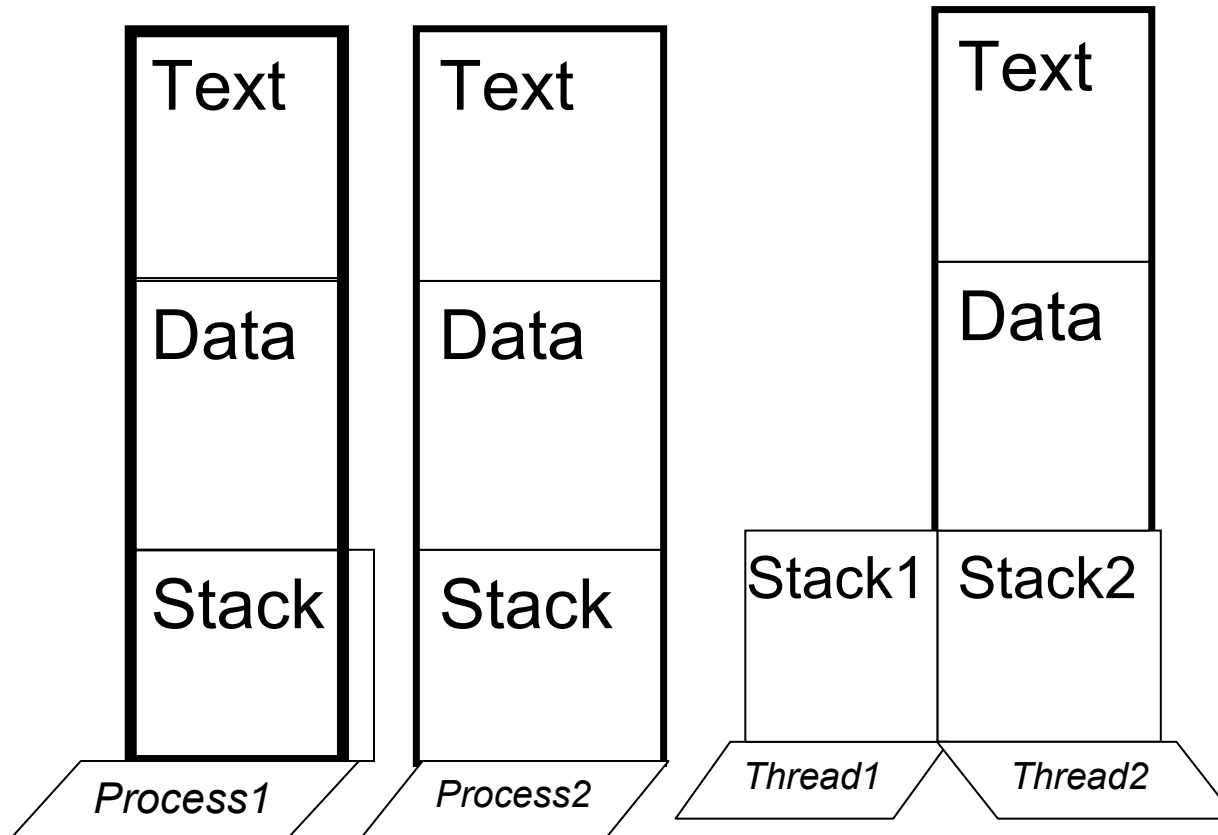


Threads (POSIX threads, pthreads)

Proces: to, čo operačný systém robí s programom

Thread: tok riadenia v procese (s vlastným stackom)



Threads (POSIX threads, pthreads)

Na čo je rozumné použiť thread:

- na vyjadrenie toho, čo sa inak vyjadriť nedá, t.j. na vyjadrenie reakcie, ktorá je nezávislá od počítania v procese (napríklad GUI). Programovanie so signálmi bolo príliš kostrbaté

Na čo nie je rozumné použiť thread:

- na všetko ostatné

Súčasné trendy použitia threadov:

- implementácia GUI
- snaha vytrieskať z PC o čosi vyšší výkon za každú cenu (web servery, numerické výpočty, hry, ...)
- budovanie špecializovaného hardwaru, ktorý profituje z použitia threadov (Intel, SGI, Nvidia/CUDA, OpenMP, ...)

Threads (POSIX threads, pthreads)

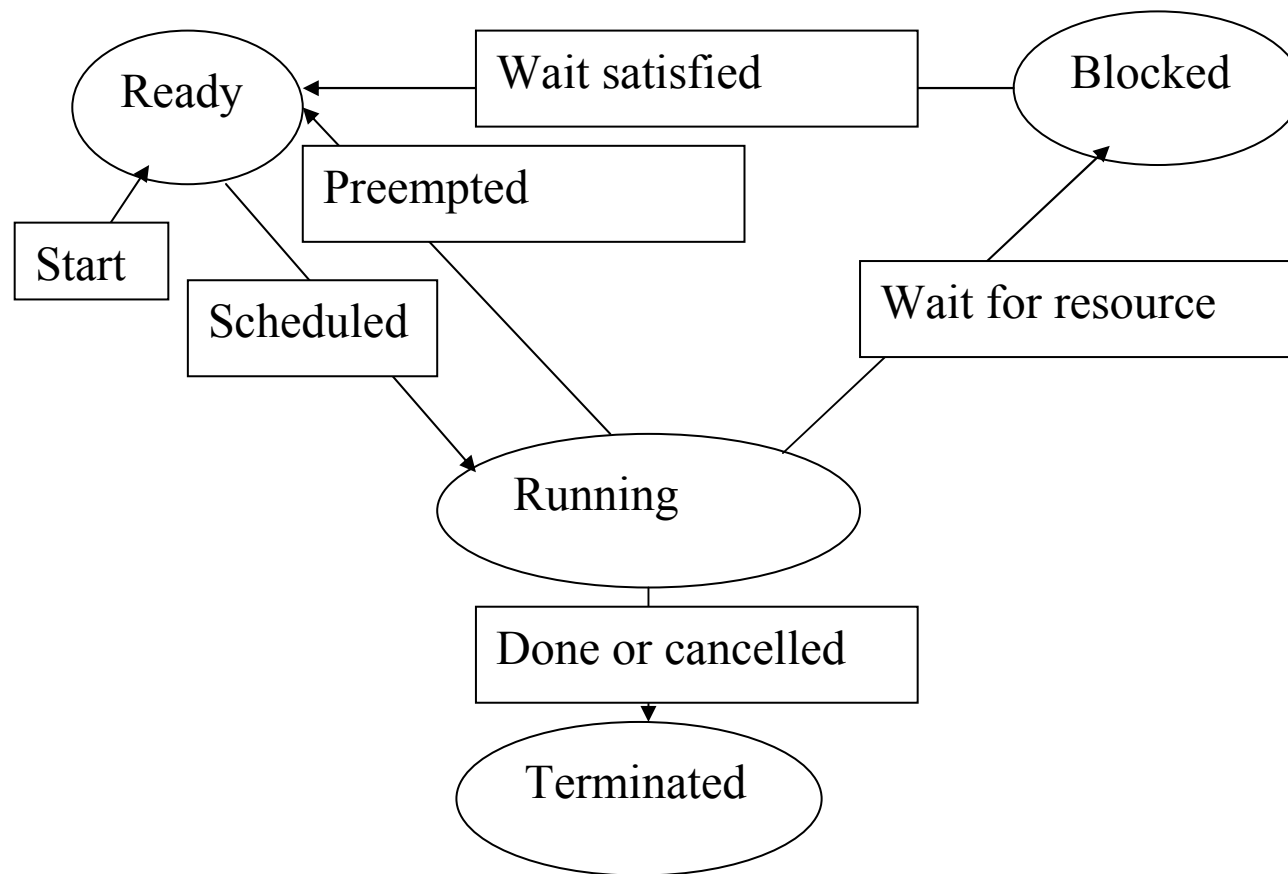
Dokonca aj na WWW sa ešte tu a tam nájdú texty, ktoré nie sú trendové (blbé). Tu je jeden z väčšej časti rozumný text o threadoch (mimochodom, skúste vygooglovať PPT prezentáciu a otvoriť na svojom PC):

“Why threads are a bad idea (for most purposes)”

John Ousterhout, Sun Microsystems Labs, 1995

Threads (POSIX threads, pthreads)

Stavový diagram threadu sa podobá na stavový diagram procesu



Threads (POSIX threads, pthreads)

API (v skutočnosti obsahuje hŕbu ďalších, zbytočných funkcií):

<code>pthread_create()</code>	vyrobí nový thread a vráti jeho id
<code>pthread_self()</code>	vráti vlastné id
<code>pthread_equal()</code>	porovná 2 ids
<code>pthread_join()</code>	čaká na ukončenie threadu
<code>pthread_mutex_init()</code>	inicializuje mutex
<code>pthread_mutex_destroy()</code>	deinicializuje mutex
<code>pthread_mutex_lock()</code>	zamkne mutex
<code>pthread_mutex_unlock()</code>	odomkne mutex
<code>pthread_cond_init()</code>	inicializuje podmienkovú premennú
<code>pthread_cond_destroy()</code>	deinicializuje podm. premennú
<code>pthread_cond_signal()</code>	signalizuje podmienkovú premennú
<code>pthread_cond_wait()</code>	čaká na podmienkovej premennej

Threads (POSIX threads, pthreads)

Príklad: pthread_create a pthread_join

```
#include <pthread.h>
```

```
#include <stdio.h>
```

```
void *thread_routine(void* arg) {  
    printf("Inside newly created thread\n");  
}
```

```
int main() {  
    pthread_t thread_id;  
    void *thread_result;  
    pthread_create( &thread_id, NULL, thread_routine, NULL );  
    printf("Inside main thread \n");  
    pthread_join(thread_id, &thread_result );  
}
```

```
gcc test.c -lpthread
```

Threads (POSIX threads, pthreads)

Predošlý program netestuje návratové hodnoty funkcií `pthread_create()` a `pthread_join()`. Na slajde to robiť netreba, v skutočnom kóde áno:

```
if ((pthread_create(&thread_id, NULL, thread_routine, NULL) != 0)
{
    ERROR("pthread_create() failed\n");
}
```

VIth Commandment

If a function be advertised to return an error code in the event of difficulties, thou shalt check that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest „it cannot happen to me“, the gods shall surely punish thee for thy arrogance.

Ten Commandments for C programmers (Henry Spencer)

1. Thou shalt run lint frequently and study its pronouncements with care, for verily its perception and judgement oft exceed thine.

2. Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.

3. Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.

4. If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.

5. Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou tighest "foo" someone someday shall type "supercalifragilisticexpialidocious".

6. If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest "it cannot happen to me", the gods shall surely punish thee for thy arrogance.

7. Thou shalt study thy libraries and strive not to re-invent them without cause, that thy code may be short and readable and thy days pleasant and productive.

8. Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.

9. Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.

10. Thou shalt forswear, renounce, and abjure the vile heresy which claimeth that "All the world's a VAX,, (or a PC, respectively), and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.

Pthreads: producer-consumer

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int shared_data =1;
void *consumer(void* arg) {
    int i;
    for (i =0; i < 30; i ++ ) {
        pthread_mutex_lock( &mutex );
        shared_data --; /* Critical Section. */
        pthread_mutex_unlock( &mutex );
    }
}
void main() {
    int i;
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, consumer, NULL);
    for(int i =0; i < 30; i ++ ) {
        pthread_mutex_lock( &mutex );
        shared_data ++; /* Producer Critical Section. */
        pthread_mutex_unlock( &mutex );
    }
    pthread_join(&thread_id);
    printf("End of main =%d\n", shared_data);
}
```

Pthreads: producer-consumer (bounded buffer)

```
#define QUEUE_SIZE 10
#define NR_ITERATIONS 1000

int in = 0; /* reading index */
int out = 0; /* writing index */
int queue[QUEUE_SIZE];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int waiting_empty = TRUE;
pthread_cond_t empty_cond = PTHREAD_COND_INITIALIZER;
int waiting_full = FALSE;
pthread_cond_t full_cond = PTHREAD_COND_INITIALIZER;

void main()
{
    pthread_t thread_consumer;
    pthread_create(&thread_consumer, NULL, consumer, NULL);
    producer();
    pthread_join(&thread_consumer);
}
```

Pthreads: producer-consumer (bounded buffer)

```
void *consumer(void* arg) {
    int i;

    for (i = 0; i < NR_ITERATIONS; i++)
    {
        pthread_mutex_lock(&mutex); /* Begin critical section */
        while (in == out) /* The queue is empty (why "while", not "if"?) */
        {
            waiting_empty = TRUE;
            pthread_cond_wait(&empty_cond, &mutex );
        }

        /* When we get here, the queue is not empty. Read from queue[in], increase in */
        in = (in + 1) % QUEUE_SIZE;
        if (waiting_full) /* Why not just cond_signal? */
            waiting_full = FALSE;
        pthread_cond_signal(&full_cond);
        pthread_mutex_unlock(&mutex); /* End critical section */
    }
}
```

Pthreads: producer-consumer (bounded buffer)

```
void producer()
{
    int i;

    for (i = 0; i < NR_ITERATIONS; i++)
    {
        pthread_mutex_lock( &mutex );
        while ((in + 1) % QUEUE_SIZE == out)
        {
            waiting_full = TRUE;
            pthread_cond_wait(&full_cond, &mutex );
        }
        /* When we get here, the queue is not empty. Write to queue[out], increase out */
        out = (out + 1) % QUEUE_SIZE;
        if (waiting_empty)
            waiting_empty = FALSE;
        pthread_cond_signal(&empty_cond );
        pthread_mutex_unlock(&mutex);
    }
}
```

Pthreads: rules of thumb

- Shared data **must** always be accessed through a single mutex
- Think of a boolean condition (expressed in terms of program variables) for each condition variable. Every time the value of the boolean condition changes, do not forget to `signal()` the condition variable
 - Call `Signal` with a locked mutex; and only call it when you are certain that another thread is waiting for the signal (**why?**)
- Avoid deadlocks
 - If possible, globally order locks, acquire them in the same order in all threads

Pthreads: Linux (contention scope)

- Contention scope is the POSIX term for describing bound and unbound threads
 - A bound thread is said to have system contention scope, i.e. it contends with all threads in the system
 - An unbound thread has process contention scope, i.e. it contends with threads in the same process

Linux uses bound threads, which is not a good idea. Think of scheduling policies. You cannot influence scheduling of processes, but you should be able to schedule your threads in any way you like. With e.g. Linux, you can not do that (there is only 1 OS which allows for that; but that one is a history)

Pthreads ponúka aj iné funkcie než tie, ktoré boli uvedené na predošlých slajdoch. Okrem toho ponúka volať ich s inými argumentami než ako bolo uvedené v príkladoch

- Doporučujem **nepoužívať** iné funkcie, narobíte menej chýb. *Concurrent programming* je dostatočne zaujímavá (t.j. netriviálna) teória, ktorá ich tiež nepoužíva
- Doporučujem **nepoužívať** iné argumenty, než ako bolo uvedené

Príklad:

```
pthread_mutex_init(pthread_mutex_t *mutex,  
pthread_mutex_attr *attr)
```

Správne použitie:

```
pthread_mutex_init(&mutex, NULL);
```

Nesprávne použitie:

akékoľvek iné

To NULL znamená default behaviour (fast mutex). Pri debuggovaní je možno rozumné použiť error checking mutex. Ale pozor na *recursive mutex*, ktorého sémantika zodpovedá Javovskému *synchronized*. Ten nechcete použiť nikdy

Ešte niekoľko tipov

- Kedykoľvek cítite potrebu použiť sleep, aby ste svoj program „rozbehali“, tak niečo robíte zle (alebo niekto vyšší)
- Kedykoľvek cítite potrebu použiť akúkoľvek formu pollingu, tak niečo robíte zle (alebo niekto vyšší). Napríklad, Petersenov algoritmus je síce korektným, ale zároveň najhorším riešením vzájomného vylúčenia. Odkedy existujú (atomické) test-and-set inštrukcie, resp. semaforey, nikto ho nechce použiť

Existuje jediná výnimka, kde je polling dokázateľne rozumným riešením: Ethernet protocol, BLAM. (Motto: tam, kde semaforey nie sú, ich treba implementovať)

Petersenov algoritmus

Petersenov algoritmus garantuje vzájomné vylúčenie dvoch threadov bez použitia semaforov. Bohužiaľ, s použitím polling a s použitím súčasného čítania zdieľaných premenných. Základná verzia:

```
int in1 = FALSE; int in2 = FALSE; int last = 1;
```

Thread 1:

```
in1 = TRUE; last = 1; /* entry protocol for thread1 */  
while (in2 && last = 1)  
    ;  
critical section  
in1 = FALSE; /* exit protocol for thread1 */
```

Thread 2:

```
in2 = TRUE; last = 2; /* entry protocol for thread1 */  
while (in2 && last = 2)  
    ;  
critical section  
in2 = FALSE; /* exit protocol for thread1 */
```

Petersenov algoritmus, ticket, bakery

Existuje zovšeobecnenie Petersenovho algoritmu pre ľubovoľný počet threadov (nie celkom jednoduché na pochopenie). Idea: každý thread musí prejsť $(N-1)$ -fázovým vstupným protokolom, kde N je počet threadov. Stačí garantovať, že práve 1 thread prejde $N-1$ fázami v jednom momente, t.j. v každej fáze stačí vyriešiť inštanciu problému vzájomného vylúčenia pre 2 thready

Existujú alternatívne riešenia, ľahšie na pochopenie: ticket algorithm, bakery algorithm

Idea ticket algoritmu: thread si pri vstupe „vezme časenku“ s číslom (ktoré sa vzápätí inkrementuje). Vstup je pre thread voľný, keď sa “na displayi“ objaví jeho číslo

Idea bakery algoritmu: thread sa pri vstupe pozrie na čísla čakajúcich threadov a zvolí si nejaké väčšie číslo. Keď toto jeho číslo začne byť najmenšie, smie vstúpiť

Petersenov algoritmus, ticket, bakery

Problém ticket algoritmu: integery sú ohraničené, po čase čísla „pretečú“

Pre bakery algoritmus toto platí tiež, ale len vtedy, keď stále nejaký thread čaká na vstup do kritického úseku

Rozumnou dodatočnou požiadavkou na mechanizmus vzájomného vylúčenia je garancia **fairness**, t.j. každý thread sa raz musí dostať do kritického úseku, keď o to požiada.
Petersenov algoritmus, ticket a bakery algoritmy sú fair

Binary logarithmic arbitrage protocol (method) rieši problém vzájomného vylúčenia za pomoci detekcie konfliktov. Používa sa v Ethernetovských sieťach, ktoré zdieľajú zbernicu (bus)

BLAM dáva “štatistickú garanciu”, že nevznikne livelock

BLAM je “štatisticky fair”—presnejšie, platí to o korektnej verzii BLAM (je zaujímavé pozrieť sa, ktorá verzia je implementovaná vo vašej sieťovej karte)

Doporučená literatúra

Literatúra:

G. Andrews: Concurrent programming, Addison-Wesley, 1989

B. Eckel: Thinking in Java, Prentice Hall, 2003

Módne trendy:

Scala Actors, <http://www.scala-lang.org/node/242>

Transactional memory, <http://clojure.org/refs>