

Operacinės sistemos

P175B304

6 paskaita

Doc. Ingrida Lagzdinytė-Budnikė

2014-03-25

Paskaitos turinys

- Procesų valdymas
 - Procesas. Jo būvis, kontekstas. Persijungimas nuo vieno proceso prie kito.
 - Gijos, realizacijos modeliai. Proceso-gijos skirtumai.
 - Procesų vykdymo planavimas. Tikslai, mechanizmai, naudojimo sąlygos.
 - **Tarprocesinė (IPC) komunikacija, klasikinės IPC komunikacijos problemos.**

Kodėl reikalinga procesų sinchronizacija?

- Paprastai sistemoje yra visa aibė bendrų resursų, kuriais lygiagrečiai veikiantys procesai/gijos nori naudotis vienu metu:
 - Failas;
 - Kintamasis
 - Spausdintuvas
 - Registras ir t.t.
- Tokie bendri resursai vadinami kritiniais resursais.
- *Kažkas ir kažkaip* turėtų užtikrinti, kad jei vienas procesas/gija konkrečiu laiko momentu naudoja atitinkamus resursus, tai kiti procesai/gijos, to padaryti negalės.

Konkurencija: Pavyzdys

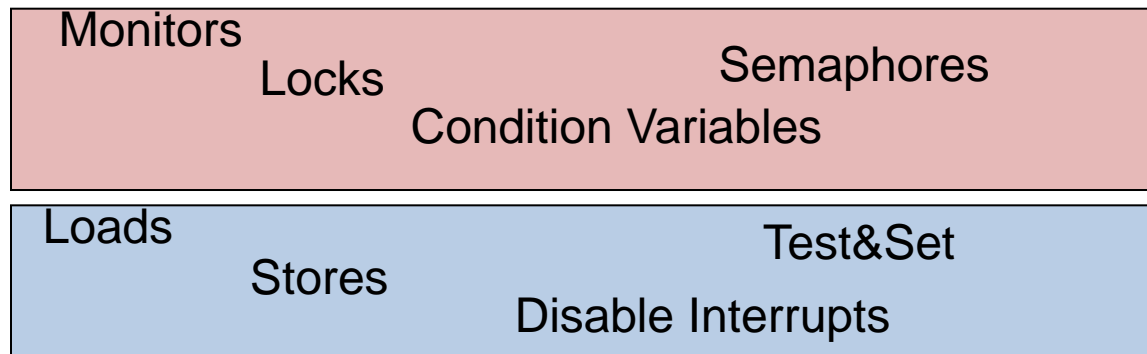
```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include "mythreads.h"
4
5  static volatile int counter = 0;
14 void *
15 mythread(void *arg)
16 {
17     printf("%s: begin\n", (char *) arg);
18     int i;
19     for (i = 0; i < 1e7; i++) {
20         counter = counter + 1;
21     }
22     printf("%s: done\n", (char *) arg);
23     return NULL;
24 }
25
32 int
33 main(int argc, char *argv[])
34 {
35     pthread_t p1, p2;
36     printf("main: begin (counter = %d)\n", counter);
37     Pthread_create(&p1, NULL, mythread, "A");
38     Pthread_create(&p2, NULL, mythread, "B");
39
40     // join waits for the threads to finish
41     Pthread_join(p1, NULL);
42     Pthread_join(p2, NULL);
43     printf("main: done with both (counter = %d)\n", counter);
44     return 0;
45 }
```

Kodėl reikalinga sinchronizacija?

- *Kažkas ir kažkaip* turėtų užtikrinti, kad jei vienas procesas/gija konkrečiu laiko momentu naudoja atitinkamus resursus, tai kiti procesai/gijos, to padaryti negalės.
- Kažkas – programuotojas, jei kuria daugiagiję/daugiaprocesę programą.
- Kažkaip:
 - Užraktai (angl. locks)
 - Semaforai (angl. semaphores)
 - Sąlygos kintamieji (angl. condition variables)
 - Monitoriai (angl. monitors)
 - Įvairūs kiti algoritmai

Procesų sinchronizaciją užtikrinančių sprendimų tipai

- **Programiniai sprendimai.**
 - realizuojami programiniais principais, naudojami algoritmai, kurie užtikrina procesų tarpusavio išskirtinumą
- **Techninės įrangos sprendimai.**
 - Jie pagrįsti tam tikrų, specialių mašininių komandų panaudojimu.
- **Operacinių sistemų sprendimai**
 - pateikiamos tam tikros operacinės sistemos **funkcijos** bei **duomenų struktūros**, leidžiančios spręsti kritinės sekcijos problemas.



Tarpusavio atskyrimas (angl. mutual exclusion)

- bet kuriuo laiko momentu tik vienas procesas gali vykdyti veiksmus su bendru (kritiniu) resursu:
 - Programos kodo dalimi;
 - Kintamuoju ir pan.

Kritinė sekcija

- Tai programos kodo dalis, kuriai turime užtikrinti tarpusavio atskyrimą:
 - bet kuriuo laiko momentu tik vienam procesui yra leidžiama atlikti kritinės sekcijos veiksmus (netgi esant keliems CPU).
 - Tik po to, kai vienas procesas atlieka visus kritinės sekcijos veiksmus, į kritinę sekciją leidžiama įeiti kitam procesui.
- Turi būti kiek įmanoma trumpesnė
- Kritinių sekcijų identifikavimas – gera sistemos/aplikacijos projektavimo pradžia

Programa su kritine sekcija

Procesas turi užsiprašyti leidimo įeiti į kritinę sekciją (CS).

Toliau seka **veiksmas** kritinėje sekcijoje,

Išėjimo sekcijos dalyje yra **pranešama** apie tai, kad procesas atlaisvino kritinę sekciją.

Likusioje sekcijoje gali būti vykdomi veiksmas, nesurišti su bendrai naudojamais kintamaisiais.

Proceso vykdomo kodo struktūra:

```
repeat
  Įėjimo sekcija
  Kritinė sekcija (CS)
  Išėjimo sekcija
  Likusi sekcija (RS)
forever
```



Programa su kritine sekcija - pavyzdys

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include "mythreads.h"
4
5  static volatile int counter = 0;
14 void *
15 mythread(void *arg)
16 {
17     printf("%s: begin\n", (char *) arg);
18     int i;
19     for (i = 0; i < 1e7; i++) {
20         counter = counter + 1;
21     }
22     printf("%s: done\n", (char *) arg);
23     return NULL;
24 }
25
32 int
33 main(int argc, char *argv[])
34 {
35     pthread_t p1, p2;
36     printf("main: begin (counter = %d)\n", counter);
37     Pthread_create(&p1, NULL, mythread, "A");
38     Pthread_create(&p2, NULL, mythread, "B");
39
40     // join waits for the threads to finish
41     Pthread_join(p1, NULL);
42     Pthread_join(p2, NULL);
43     printf("main: done with both (counter = %d)\n", counter);
44     return 0;
45 }
```

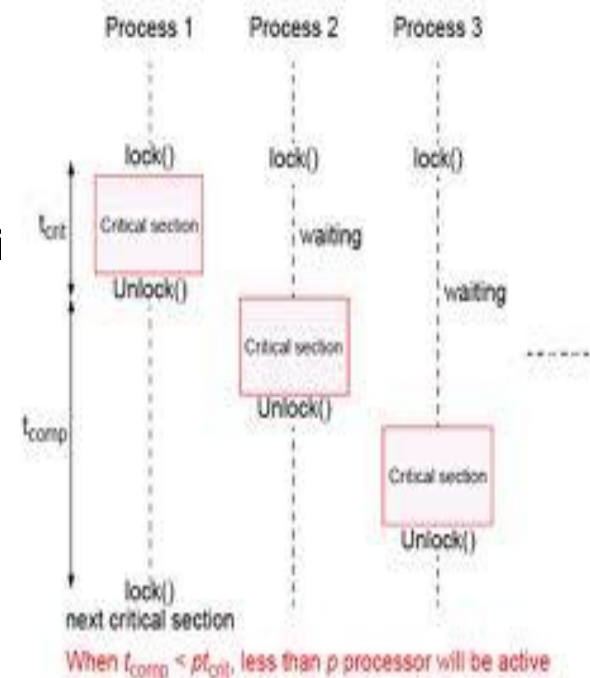
Įėjimo sekcija

Kritinė sekcija

Išėjimo sekcija

Reikalavimai efektyviam kritinės sekcijos problemos sprendimui

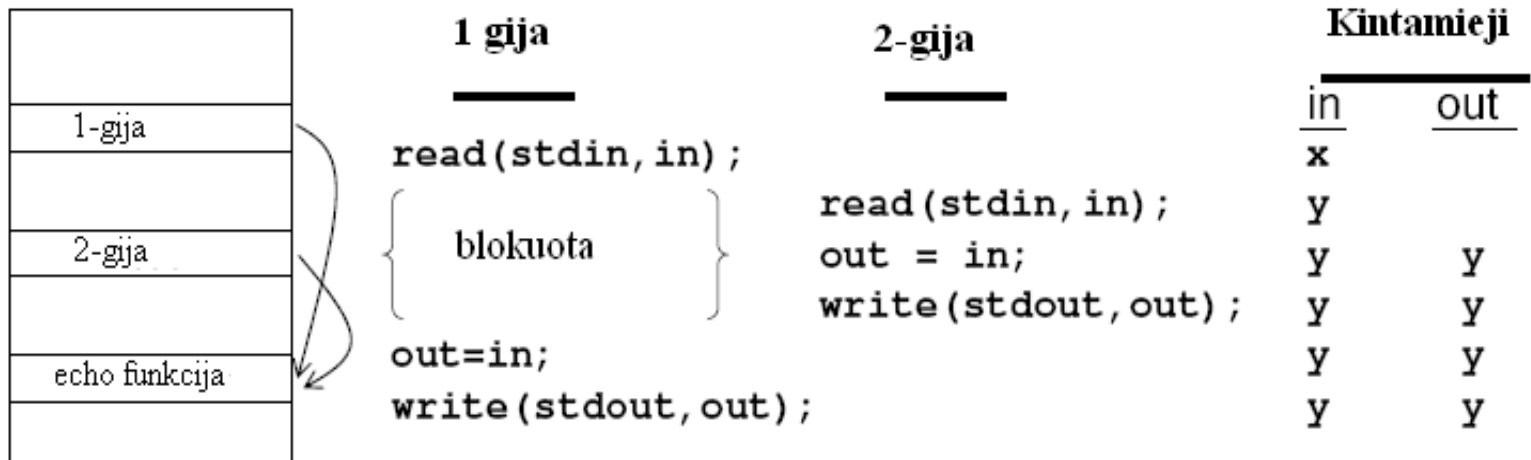
- **Tarpusavio atskyrimo reikalavimas:**
 - bet kuriuo laiko momentu tik vienas procesas gali vykdyti kritinės sekcijos (CS) veiksmus.
- **Eigos-progreso reikalavimas:**
 - Tik tie procesai, kurie yra „įėjimo“ sekcijoje gali įeiti į savas CS. Šis įėjimas negali būti atidėtas neapibrėžtam laikui.
- **Ribinio laukimo reikalavimas:**
 - Po to, kai procesas užsiprašo leidimo įeiti į kritinę sekciją (CS), egzistuoja tam tikra ribinė laukimo reikšmė, kuri nusako kiek kartų kitiems procesams bus leista įeiti į savas CS iki to momento, kol šis procesas įeis į CS



Lenktynių situacija (angl. race condition)

- Tai situacija, kai programos įvykdymo rezultatas priklauso nuo procesų (gijų) vykdymo sekos.

```
void echo (char in)
{
    char out;
    read (stdin, in);
    out = in;
    write (stdout, out);
}
```



Lenktynių situacija: Pavyzdys

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include "mythreads.h"
4
5  static volatile int counter = 0;
14 void *
15 mythread(void *arg)
16 {
17     printf("%s: begin\n", (char *) arg);
18     int i;
19     for (i = 0; i < 1e7; i++) {
20         counter = counter + 1;
21     }
22     printf("%s: done\n", (char *) arg);
23     return NULL;
24 }
25
32 int
33 main(int argc, char *argv[])
34 {
35     pthread_t p1, p2;
36     printf("main: begin (counter = %d)\n", counter);
37     Pthread_create(&p1, NULL, mythread, "A");
38     Pthread_create(&p2, NULL, mythread, "B");
39
40     // join waits for the threads to finish
41     Pthread_join(p1, NULL);
42     Pthread_join(p2, NULL);
43     printf("main: done with both (counter = %d)\n", counter);
44     return 0;
45 }
```

Lenktynių situacija: Pavyzdys

- 1 bandymas

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

- 2 bandymas

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

- 3 bandymas

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Lenktynių situacija: Pavyzdys

- `counter` kintamojo, kuris saugomas adresu `0x8049a1c` reikšmės padidinimas vienetu mašininėmis komandomis:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Lenktynių situacija: Pavyzdys

- Neapibrėžtų rezultatų gražinimo priežastis – OS planuotojas:

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	<i>mov 0x8049a1c, %eax</i>		105	50	50
	<i>add \$0x1, %eax</i>		108	51	50
interrupt	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		<i>mov 0x8049a1c, %eax</i>	105	50	50
		<i>add \$0x1, %eax</i>	108	51	50
		<i>mov %eax, 0x8049a1c</i>	113	51	51
interrupt	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	50
	<i>mov %eax, 0x8049a1c</i>		113	51	51

Lenktynių situacija: Pavyzdys

- Neapibrėžtų rezultatų gražinimo priežastis – OS planuotojas:

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	<i>mov 0x8049a1c, %eax</i>		105	50	50
	<i>add \$0x1, %eax</i>		108	51	50
interrupt	<i>save T1's state</i>				
	<i>restore T2's state</i>				
		<i>mov 0x8049a1c, %eax</i>	100	0	50
		<i>add \$0x1, %eax</i>	105	50	50
		<i>mov %eax, 0x8049a1c</i>	108	51	50
interrupt			113	51	51
	<i>save T2's state</i>				
	<i>restore T1's state</i>				
	<i>mov %eax, 0x8049a1c</i>		108	51	50
			113	51	51

Lenktynių situacija: Pavyzdys

- Neapibrėžtų rezultatų gražinimo priežastis – OS planuotojas:

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	<i>mov 0x8049a1c, %eax</i>		105	50	50
	<i>add \$0x1, %eax</i>		108	51	50
interrupt	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		<i>mov 0x8049a1c, %eax</i>	105	50	50
		<i>add \$0x1, %eax</i>	108	51	50
		<i>mov %eax, 0x8049a1c</i>	113	51	51
interrupt	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	50
	<i>mov %eax, 0x8049a1c</i>		113	51	51

Lenktynių situacija: Pavyzdys

- Neapibrėžtų rezultatų gražinimo priežastis – OS planuotojas:

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	<i>mov 0x8049a1c, %eax</i>		105	50	50
	<i>add \$0x1, %eax</i>		108	51	50
interrupt	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		<i>mov 0x8049a1c, %eax</i>	105	50	50
		<i>add \$0x1, %eax</i>	108	51	50
		<i>mov %eax, 0x8049a1c</i>	113	51	51
interrupt	<i>save T2's state</i>				
	<i>restore T1's state</i>				
	<i>mov %eax, 0x8049a1c</i>				

108	51	50
113	51	51

Vietoje 52 gauname 51 !

Praktika lenktynių situacijų mažinimui

- Tai pačiai kritinei sekcijai naudoti tą patį sinchronizacijos mechanizmą.
- Jei tik yra galimybė, naudoti tuos sinchronizavimo mechanizmus, kuriuos pateikia konkreti API (nekurt jų patiems).
- Dokumentuoti, kokie sinchronizacijos mechanizmai kokiems sistemos resursams skirti (t.y. kokio resurso prieiga yra sinchronizuojama)
- Jei darbas komandinis – duoti savo programos dalis tikrinti kitiems savo komandos draugams

Mirties taško situacija (angl. deadlock)

- Tai lenktynių situacijos priešingybė
- Situacija kai procesų arba gijų grupė blokuojasi taip, kad toliau nei vienas iš šios grupės nebegali vykdyti tolesnių veiksmų
- Mirties taškas kyla kai du ar daugiau procesų turi konfliktuojančius poreikius.
- Sąlygos mirties taško situacijai susidaryti:
 - Kritinis resursas negali būti perimamas;
 - Kritinis resursas reikalauja tarpusavio išskirtinumo užtikrinimo;
 - Vienų kritinių resursų laukiantys užsiblokavę procesai arba gijos gali dar kartą užsiblokuoti prieigai prie kitų kritinių resursų gauti;
 - Egzistuoja ciklas grafe, kurio viršūnės identifikuoja procesus arba gijas, kurie laukia vieni kitų įvykdymo pabaigos

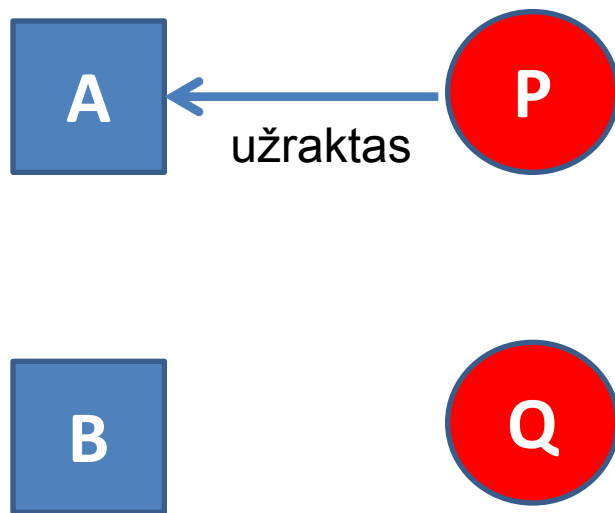
Mirties taško situacijos grafinė iliustracija

- Įsivaizduokime, kad turime dvi gijas (P ir Q) ir du objektus (A ir B) - kritinius resursus



Mirties taško situacijos grafinė iliustracija

- Įsivaizduokime, kad turime dvi gijas (P ir Q) ir du objektus (A ir B) - kritinius resursus



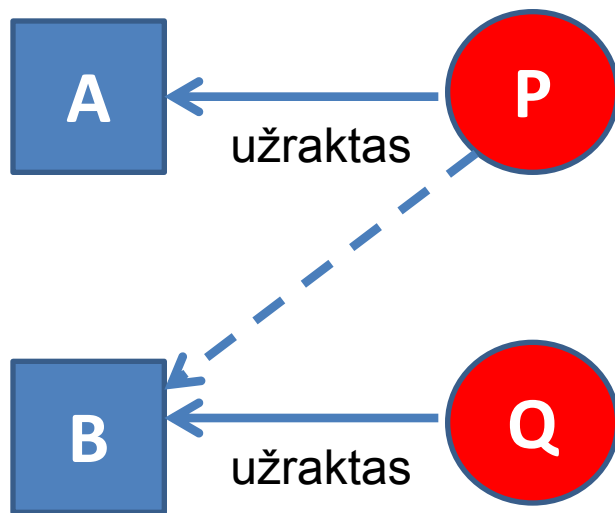
Mirties taško situacijos grafinė iliustracija

- Įsivaizduokime, kad turime dvi gijas (P ir Q) ir du objektus (A ir B) - kritinius resursus



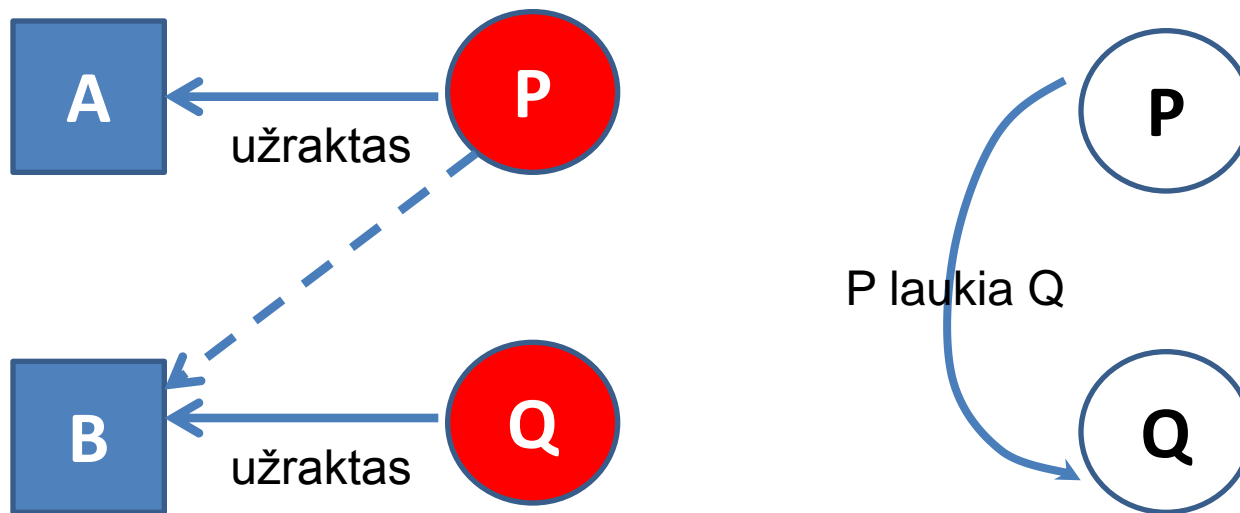
Mirties taško situacijos grafinė iliustracija

- Įsivaizduokime, kad turime dvi gijas (P ir Q) ir du objektus (A ir B) - kritinius resursus



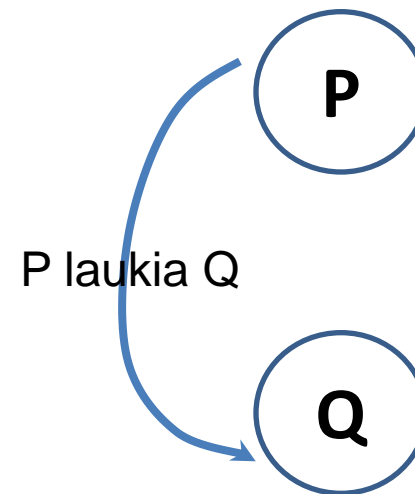
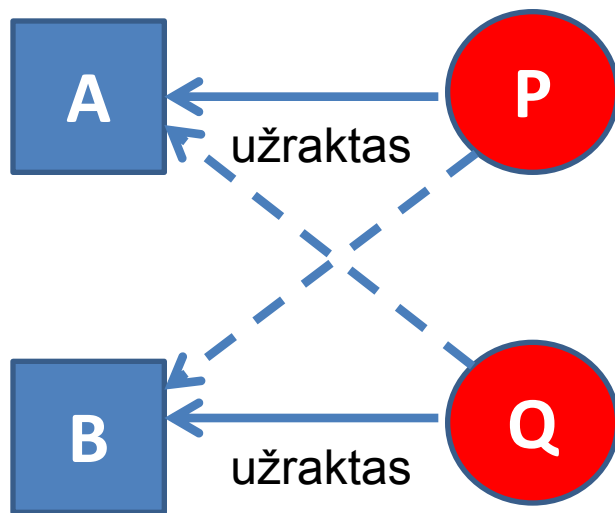
Mirties taško situacijos grafinė iliustracija

- Įsivaizduokime, kad turime dvi gijas (P ir Q) ir du objektus (A ir B) - kritinius resursus



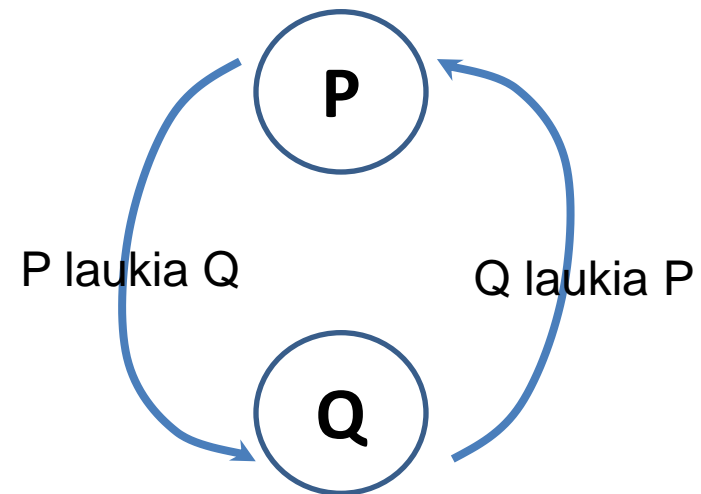
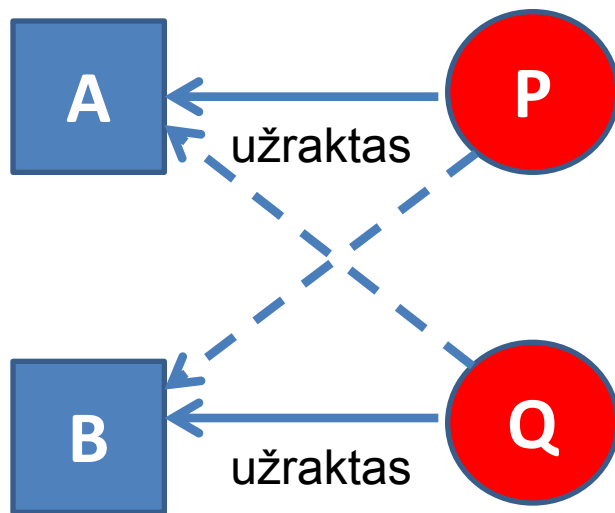
Mirties taško situacijos grafinė iliustracija

- Įsivaizduokime, kad turime dvi gijas (P ir Q) ir du objektus (A ir B) - kritinius resursus



Mirties taško situacijos grafinė iliustracija

- Įsivaizduokime, kad turime dvi gijas (P ir Q) ir du objektus (A ir B) - kritinius resursus



Kaip išvengti mirties taško?

- Egzistuoja 2 galimybės
 - Sutrukdyti mirties taškui susidaryti (t.y. vengti mirties taško susidarymo – vykdyti *mirties taško prevenciją*)
 - Leisti mirties taškui susidaryti, nustatyti, kad jis susidarė ir jį nutraukti (t.y. *leisti* mirties taško situacijai *susidaryti ir realizuoti būdus kaip iš jo išeiti*).

Mirties taško prevencija

- Procesas arba gija vienu laiko momentu neturėtų reikalauti daugiau negu vieno resurso (nelankstu)
- Reikalauti resursų ta pačia tvarka (ne visada įvykdomas, nes ne visuomet žinoma kokių resursų reikės)
- Prieš procesui ar gijai pereinant į blokuotą būseną, įvertinti mirties taško susidarymo galimybes ir jei jos realios, nutraukti veiksmų vykdymą apskritai (gali būti prarandama nemaža dalis jau atliktų darbų)

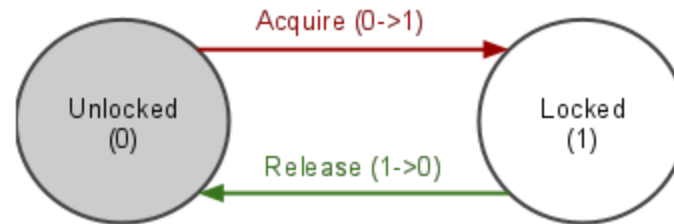
Badavimo situacija (angl. starvation)

- Tai situacija, kai vienas ar daugiau procesų (gijų) laukia kritinių resursų, bet jų negauna.
- Kada tokia situacija gali susidaryti?
 - Kai procesų/gijų vykdymo planavimo procesas nedeterminuotas (aiškiai neapibrėžtas)
 - Planavimo algoritmas toks, kad didesnio prioriteto procesai vykdomi pirmi.

Procesų sinchronizacijos mechanizmai

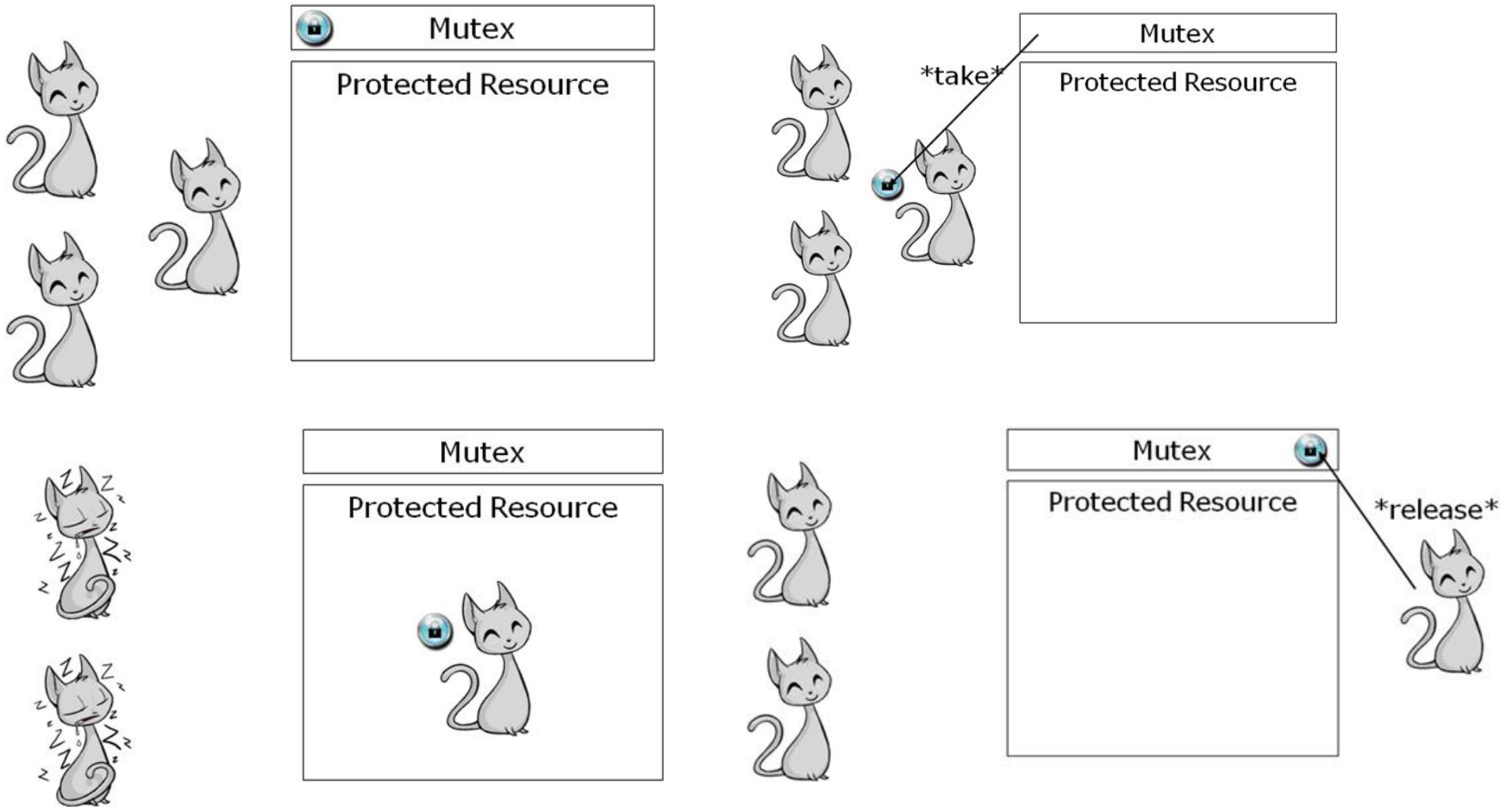
UŽRAKTAI (angl. locks)

Užraktų naudojimo tikslas, jų savybės



- naudojami *tik procesų tarpusavio atskyrimo užtikrinimui*, valdant prieigą prie bendrai naudojamų duomenų ar bendros kodo dalies:
- gali būti viename iš dviejų būvių:
 - užrakintas arba atrakintas (1-0).
- Sudaro 3 pagrindinės operacijos:
 - Allocate and Initialize – užrakto inicializacija
 - Acquire /lock – gauti prieigą prie resurso
 - Release /unlock – atlaisvinti prieigą prie resurso

Užraktų naudojimo logika



Užraktų naudojimo pavyzdžiai (1)

Kritinė sekcija:

```
balance = balance + 1;
```

Kritinės sekcijos apsauga užraktais:

```
1  lock_t mutex; // some globally-allocated lock 'mutex'
2  ...
3  lock(&mutex);
4  balance = balance + 1;
5  unlock(&mutex);
```

Kritinės sekcijos apsauga užraktais naudojant *POSIX Pthread* biblioteką:

```
1  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
4  balance = balance + 1;
5  Pthread_mutex_unlock(&lock);
```

Užraktų naudojimo pavyzdžiai (2)

- Po užrakto inicializacijos

```
Void deposit(int amount) {  
    Pthread_mutex_lock(&mylock);  
    balance += amount;  
    Pthread_mutex_unlock(&mylock);  
}
```

- Išskirti po vieną užraktą kiekvienai banko sąskaitai

```
Void deposit(int accountid, int amount) {  
    Pthread_mutex_lock(&locks[accountid]);  
    balance[accountid] += amount;  
    Pthread_mutex_unlock(&locks[accountid]);  
}
```

Metrikos užraktams (procesų sinchronizaciją užtikrinantiems mechanizmams) vertinti

- Tarpusavio atskyrimas (veikia, neveikia)
- Sąžiningumas
- Efektyvumas (papildomos laiko sąnaudos, susidariusios dėl užraktų naudojimo):
 - Kai turime 1 procesorių;
 - Kai turime 2 ir daugiau procesorių.

Užraktų realizacija (1)

- Kritinės sekcijos vykdymo metu uždraudžiant pertrauktis arba keičiant jų vykdymo prioritetą:
 - Trumpalaikio planuotojo-dispečerio veikla laikinai pristabdyta;
 - Kritinės sekcijos veiksmai vykdomi nepertraukimai – kaip atominis vienetas

```
1 void lock() {  
2     DisableInterrupts();  
3 }  
4 void unlock() {  
5     EnableInterrupts();  
6 }
```

Privalumai: paprastas.

Trūkumai?

Užraktų realizacija (2)

- Naudojant žemo lygio komandas, kurių atomiškumas užtikrinamas techninės įrangos pagalba:
 - Test &Set (TAS)
 - Load&Store (LL/SC)
 - Compare&Swap (CAS)
 - xchg (a,b) ir t.t.

Užraktų realizacija (3)

- Naudojant žemo lygio komandas, kurių atomiškumas užtikrinamas techninės įrangos pagalba:
- TAS: Test and Set `addr val` – grąžina `addr` reikšmę ir ją nustato į `val`.

Pavyzdys: `C=10`

`Old = TAS(&C, 15)`

`Old == 10 C == 15`

```
1      int TestAndSet(int *ptr, int new) {
2          int old = *ptr; // fetch old value at ptr
3          *ptr = new;      // store 'new' into ptr
4          return old;      // return the old value
5      }
```


Užraktų realizacija (4)

- Naudojant TestAndSet()


```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available, 1 that it is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Užraktų realizacija (5)

Spinlock (realizuoto TestAndSet pagrindu) įvertinimas:

- Tarpusavio atskyrimas - užtikrinamas

- Sąžiningumas - neužtikrinamas



Realizuojant užraktą *Fetch&Add* komandos pagalba galima realizuoti prieigos prie kritinės sekcijos valdymą – t.y. fiksuoti kokia gija jau gavo prieigą, o kokia ne, ir tokiu būdu išvengti badavimo situacijos. Paanalizuokite kaip *Fetch&Add* leidžia tai užtikrinti

- Efektyvumas

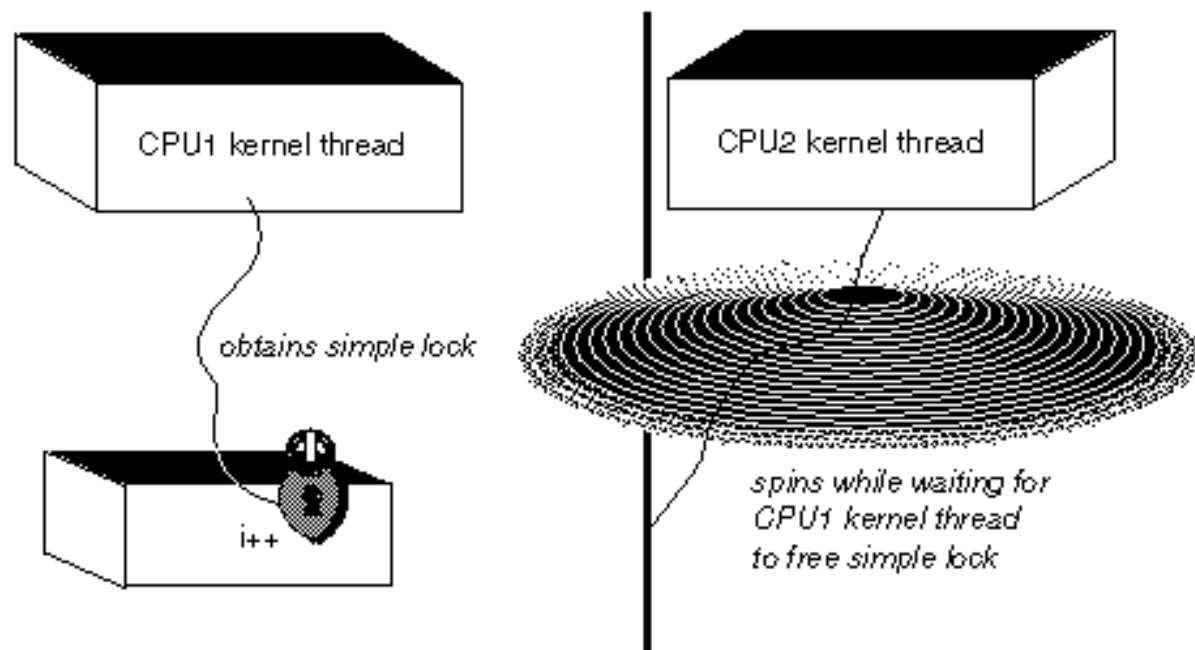
Užraktų realizacija (5)

Spinlock (realizuoto TestAndSet pagrindu) įvertinimas:

- Tarpusavio atskyrimas - užtikrinamas
- Sąžiningumas - neužtikrinamas
- Efektyvumas –
 - 1) esant vienam procesoriui – naudoti ne visada efektyvu
 - 2) esant daugeliui procesorių – naudoti efektyvu.

Pagalvokite kodėl?

Ciklinis laukimas ir gijų blokavimas



ZK-0957U-AI

Užraktų realizacija, kai CPU resursai, negavus prieigos prie kritinės sekcijos, yra atlaisvinami

- Reikalingas palaikymas iš OS pusės – sisteminiai kreipiniai, leidžiantys nutraukti gijų vykdymą. Šiame pavyzdyje panaudotas `yield()` kreipinys.

```
1  void init() {
2      flag = 0;
3  }
4
5  void lock() {
6      while (TestAndSet(&flag, 1) == 1)
7          yield(); // give up the CPU
8  }
9
10 void unlock() {
11     flag = 0;
12 }
```

Užraktų realizacija, kai negavus prieigos prie kritinės sekcijos, naudojamos eilės

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

- Gijų “užmigdymui” ir “pažaditimui” naudojami papildomi OS sisteminiai kreipiniai. (Šiuo atveju `park()` ir `unpark(ThreadID)`)

Skirtingos OS, skirtingos užraktų realizacijos

- Užraktų realizavimui skirtingose OS naudojami skirtingi sisteminiai kreipiniai (pavadinimai skiriasi, funkcionalumas gali taip pat, tačiau nežymiai)
- **Linux OS, `lowlevellock.h`, `nptl` biblioteka:**
`futex_wait(address, expected);`
`futex_wait(address);`

Dviejų lygių užraktai

- Mišraus tipo užraktas (apimantis abi aptartas realizacijas):
 - 1 fazė: tam tikrą nustatytą laiko tarpą cikliška tikrinama užrakto būseną (laisvas/nelaisvas)
 - 2 fazė: Viršijus 1-oj fazėj nustatytą laiko tarpą, gija “užmigdoma” ir pereinama prie kitos vykdymo.

Šaltiniai besidomintiems giliau:

- ✓ glibc 2.9 (include Linux pthreads implementation)“.

Prieiga per internetą: <http://ftp.gnu.org/gnu/glibc/>

Detalesnės info apie `pthread` ieškoti `nptl` kataloge.

- ✓ “OpenSolaris Thread Library”.

Prieiga per internetą: <http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/port/threads/synch.c>

Procesų sinchronizacijos mechanizmai

SEMAFORAI (angl. semaphores)

Savarankiškos studijos

- Perskaitykite ir išstudijuokite sekančią medžiagą iš šios paskaitos papildomų šaltinių:
 - Skaidrių komplekto “kritinė_sekcija.pdf” 24 – 39 skaidrės.
 - <http://pages.cs.wisc.edu/~remzi/OSTEP/> “Concurrency” skyriaus “Semaphores” poskyris
 - Jei turite knygą N. Sarafinienė “Operacinės sistemos” – 6.4 skyrius
- Studijuodami medžiagą atsakykite į sekančius klausimus:
 - Kas yra semaforas? Kuo jis skiriasi nuo užrakto?
 - Semaforų tipai. Jų privalumai ir trūkumai.
 - Kaip panaudoti semaforus praktiškai?
 - Vartotojo-gamintojo, skaitytojų-rašytojų problemos.
 - Pietaujančių filosofų problema.

Ačiū už jūsu dėmesį