Tema:
  1) Kapitola 4 knihy o xv6: “Traps and device drivers”
     hw si vynucuje pozornost,
     vykonavanie sw sa musi prerusit!
  2) Kapitola 5 knihy o xv6: “Locking”
     zamky

****

Preco si hw vyzaduje pozornost?
  1) MMU nedokaze prelozit VA --> PA (napr. PTE_V bit nie je nastaveny)
  2) uzivatelsky program deli nulou
  3) uzivatelsky program sa pokusa vykonavat kod jadra
  4) sietova karta prijala udaje, ktore treba spracovat
  5) casovac chce zvysit pocitadlo hodin
  6) je potrebne vyriesit komunikaciu medzi procesormi (napr. pri vyprazdneni TLB)

Rozlisujeme 3 druhy vynimocnych stavov:
  1) vynimky (vypadok stranky, delenie nulou...)
  2) systemove volania (ecall)
  3) prerusenia (hw zariadenia)

Terminologia v literature nie je ohladom tychto stavov jednotna. Napriklad co sa tyka RISC-V,
vsetky 3 druhy sa obsluhuju uplne rovnakou technikou, a preto sa aj casto rovnako oznacuju,
napr. tzv. traps.

Pre blizsie ozrejmenie sa zameriame na vstup z klavesnice. Ako funguje take napisanie
$ ls
  1) klavesnica posle udaje hardveru UART (universal asynchronous receiver/transmitter)
  2) UART vygeneruje prerusenie
  3) na prerusenie reaguje kod v jadre (ovladac), ktory precita udaje z UART (1 znak)
  4) pomocou dalsieho ovladaca sa tento znak vypise na konzole

Ako kernel rozpozna, ktore zariadenie vyslalo signal prerusenia?
  * kazde zariadenie ma vlastne jedinecne IRQ (interrupt request) cislo
  * toto cislo je definovane vyrobcom hardveru
  * napr. UART0 ma cislo 10 v qemu (vid kernel/memlayout.h)

Registre RISC-V suvisiace s preruseniami>
  * sie   -- supervisor interrupt enabled register
            -- 1 bit pre zapnutie preruseni
  * sstatus -- supervisor status register
            -- bity rozlisujuce zdroj prerusenia: softverove, externe, od casovaca
  * scause -- supervisor cause register
            -- dovod (pricina) prerusenia
  * stvec   -- supervisor trap vector register
            -- adresa obsluznej funkcie vynimocneho stavu

Ako funguje proces preruseni v xv6
  main():
    consoleinit():
      uartinit()
    plicinit()
    scheduler():
      intr_on():
        w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE)
        w_sstatus(r_sstatus() | SSTATUS_SIE)

```
  $
  shell je v systemovom volani read; caka na vstup z konzoly; sled volani:
  usertrap() pre systemove volanie:
    w_stvec((uint64)kernelvec)
    consoleread():
      sleep():
        scheduler():
          intr_on()
  pozor! prerusenia su zapnute!


  $ l
  pouzivatel stlacil klavesu 'l', ktora sposobi prerusenie na zbernici UART
  subsystem PLIC doruci prerusenie na procesor
  procesor vykona nasledovne kroky:
  1) ak ide o prerusenie zo zariadenia a SIE bit je nulovy, nasledovne kroky sa NEkonaju
  2) !!! vypne prerusenia vynulovanim SIE bitu !!! (preco?)
  3) skopiruje pc do sepc
  4) ulozi aktualny rezim cpu (user alebo supervisor) do SPP bitu v registri sstatus
  5) ulozi pricinu prerusenia do scause registra
  6) zmeni mod cpu na supervisor
  7) skopiruje stvec do pc (v pripade xv6 je v stvec ulozena adresa funkcie kernelvec)
  8) pokracuje vykonavanim instrukcii podla pc registra



co robi kernelvec()?
  1) alokuje miesto na aktualnom zasobniku; o ktory zasobnik ide?
     ulozi registre cpu do tohto miesta na zasobniku
     v pripade xv6 sa jedna o zasobnik planovaca
  2) kerneltrap()
       devintr()
         uartintr()
           c = uartgetc()
           consoleintr(c)
             handle ctrl characters
             echo c
             put c in buffer
             wakeup reader
       return from devintr
     return from kerneltrap
  3) nahraj hodnoty registrov zo zasobnika
  4) sret



ako sa spracuva prerusenie z uzivatelskeho priestoru? napr. sekvencia: usertests&\nls\n
  usertrap()
    ...
    w_stvec((uint64)kernelvec)
    ...
    if(r_scause() == 8){
      ...
    } else if ((which_dev = devintr()) != 0){
      // ok
    }
    ...
    usertrapret()
```

```
  usertrapret()
    intr_off()
    ...
    // spracovanie syscall, interrupt, exception pojde cez trampoline.S
    w_stvec(TRAMPOLINE + (uservec - trampoline))
    ...
    unsigned long x = r_sstatus()
    x &= ~SSTATUS_SPP // zrus supervisor mode (supervisor previous privilege mode)
    x |= SSTATUS_SPIE // zapni prerusenia v user mode
    w_sstatus(x)
    ...

  usertrapret/uservec
    rovnaky kod ako pre systemove volania!
    tento kod sa pouziva spolocne pre vsetky typy: vynimky, prerusenia, systemove volania

prerusenia zavadzaju koncep konkurentneho vykonavania kodu! (nie paralelneho!!!)
  pocas mojho (uzivatelskeho) kodu bude bezat iny kod (spracovanie prerusenia)
  priklad
    1) addi sp, sp, -48
    2) sd ra, 40(sp)
  medzi riadkami 1 a 2 sa moze vykonat iny kod! kod spracovania hw prerusenia!
  nemoze nastat kod spracovania systemoveho volania (kedze sa nepouzila instrukcia ecall)
  moze nastat kod spracovania vynimky (nenamapovana stranka s instrukciami...)
  dajme tomu, ze medzi 1) a 2) nastane prerusenie uart alebo od casovaca
  pre uzivatelsky kod je to v pohode
    kernel obnovi stav vykonavania uzivatelskeho kodu pri navrate do user priestoru
    t.j. obnovi stav procesora, cize hodnoty vsetkych registrov
  pre kod jadra je situacia tazsia!
    obsluha prerusenia moze aktualizovat udaje, ktore sa vyuzivaju na inom mieste!
    kod na inom mieste:            obsluha prerusenia:
    1  x = 0
    2  if x == 0:            x = 1
    3    f()
    takato konstrukcia znamena, ze f() sa bud vykona, alebo nevykona ;)
    to podla toho, kedy pride k obsluhe prerusenia! skusme vsetky moznosti ;)
  aby sa nejaky blok kodu vykonal "atomicky" (t.j. neprerusitelne vzhladom na prerusenia),
    prerusenia musia byt vypnute!
    intr_off():
      w_sstatus(r_sstatus() & ~SSTATUS_SIE)
  RISC-V vypina prerusenia pri prichode vsetkych typov (vynimka, sys volanie, prerusenie)
    !!! otazka: moze prist ku preruseniam v trampoline.S? !!!
```

****

hash tabulka, viacvlaknovy pristup
  paralelne operacie put() a get()
  kolizie riesene obycajnym retazenim

  struct entry{
    int key, value;
    struct entry *next;
  }

  predpokladajme put(5) a put(10) sucasne
  obe vlakna sa pokusia o zapis napr. do prvku table[0], ale v akom poradi?
    race condition (subeh):
      - ked iste poradie operacii moze sposobit nekorektne spravanie
      - sucasny pristup (aspon jeden z pristupov je zapis)

  kam umiestnit zamok?
    1) na celu tabulku? --- tzv. big lock
    2) na polozku tabulky?
    3) v kazdej strukture 'entry' na ochranu polozky 'next'?

co je to vobec zamok (lock)?
  je to objekt (abstraktny datovy typ - ADT)
  pomocou roznych objektov je mozne riadit pristup k roznym udajom
  pouzitie:
    lock l;
    acquire(&l);
    // critical section
    // i.e. x = x + 1;
    release(&l);
  viacero vlakien moze sucasne vyvolat funkciu acquire(&l):
    iba jedinemu vlaknu sa podari okamzite funkciu acquire dokoncit
    ostatne vlakna musia cakat, pokym nebude vyvolana funkcia release(&l)

na co je dobry zamok:
  1) aby vlakna/procesy po jednom pristupovali k zdielanej pamati
  2) aby sa z neatomickych operacii stala atomicka!

1) kam umiestnit zamok (z pohladu riadenia pristupu k udajom)? podla nasledovneho pravidla:
  1) mozu sucasne dve alebo viac vlakien pristupovat k nejakej pamati?
  2) je aspon jeden z tychto pristupov zapis?
  --> ak ano, NUTNE POUZIT ZAMOK

2) kam umiestnit zamok (z pohladu riadenia vykonavania kodu)?
  uviaznutie (deadlock):
    proces A: proces B:
      lock(&lock1)   lock(&lock2)
      ...   ...
      lock(&lock2)   lock(&lock1)
      ...   ...
    ani jeden z procesov nemoze pokracovat!
    riesenie:
      napr. zaviest vzdy rovnake poradie ziskavania zamkov
      to vyzaduje nasledovne operacie: predikcia, utriedenie, ziskanie zamkov
      prilis komplexne!

kam umiestnit zamok (z pohladu riadenia vykonavania kodu)? priklad dobrej strategie:
1) napisat modul (nezavislu cast programu) korektne vzhladom na seriove vykonavanie
t.j. vzhladom na vykonavanie jednym jadrom
2) POTOM pridat zamky na VYNUTENIE serializacie tam, kde moze prichadzat k subehom
kazdu takto uzamknutu cast kodu moze v jednom case vykonavat iba jedno CPU,
takze tuto cast mozeme povazovat za atomicky vykonavanu (neprerusitelna operacia)

preco pouzivat tuto strategiu?
lahsie sa nam uvazuje nad seriovo vykonavanym kodom...

aky je vztah medzi zamkami a paralelizmom?
- zamky ZABRANUJU paralelnemu vykonavaniu!!!
- ked chceme zvysit paralelizmus, zvacsa musime rozdelit udaje, zaviest viac
zamkov (pre kazdu sekciu udajov), t.j. zjemnit granularitu uzamykania
- zvolit spravny pomer medzi rozdelenim udajov a zamkami patri medzi
najvacsie vyzvy tvorby a navrhu programov:
a) cela tabulka; kazdy riadok zvlast; kazda polozka
b) cely suborovy system; kazdy adresar/subor zvlast; kazdy blok disku zvlast
c) cele jadro OS; kazdy subsystem zvlast; kazdy objekt zvlast
- niekedy je nutne pozmenit znacnu cast navrhu, aby bolo mozne zaviest
paralelizmus! toto uz moze byt veeeelmi narocna uloha!
priklad: ak vlakna cakaju na ziskanie volneho bloku pamate z jedneho zoznamu,
rozdelit jeden zoznam na zoznamy volnych blokov pamate zvlast pre kazde CPU

volba granularity uzamykania:
treba zacat s tzv. velkym uzamykanim (big locks): jeden zamok na jeden modul:
mensia sanca uviaznutia, pretoze bude mensia sanca, ze proces potrebuje dva zamky sucasne,
ked riesi nejaku samostatnu ulohu (a cele riesenie ulohy sa nachadza v jednom module)
potom je potrebne merat efektivitu riesenia a pozorovat spravanie systemu
v mnohych pripadoch staci granularita 'big lock', zvacsa sa znizi efektivita kvoli cakaniu
na jemnejsiu granularitu sa prechadza iba vtedy, ked efektivita riesenia s velkym zamkom
nestaci!

implementacia zamku:

```
struct lock { int locked; }
acquire(lock *l){
  while(1){
    if(l->locked == 0){ // riadok A
      l->locked = 1;    // riadok B
      return;
    }
  }
}
```

chybaaaa!!! subeh (race condition) medzi riadkami A a B!
je mozne nejakym sposobom vykonat oba riadky atomicky?

softverovo veeeelmi tazko, vyzaduje sa nutna podpora hw; instrukcia exchange:

```
mov $1, %eax    // daj do registra eax hodnotu 1
xchg %eax, addr // atomicky vymen obsah registra eax s hodnotou na adrese addr
```

co vykona hardver pri instrukcii xchg?
1) lock addr globalne (t.j. pre vsetky CPU a jadra uzamkne pristup k addr)
2) temp <-- *addr
3) *addr <-- %eax
4) %eax <-- temp
5) unlock addr
takto je to riesene na x86 (hw uzamknutim pamatovej oblasti)
rozny hw to riesi roznym sposobom

!!! implementacia zamku s instrukciou xchg: !!!

```
struct lock { int locked; }
acquire(lock *l){
  while(1){
    if(xchg(&l->locked, 1) == 0)
      break;
  }
}
```

analyza kodu:

1) ak je uz hodnota l->locked 1, xchg ju (znovu) nastavi na 1, vrati 1, a pokracuje dalsim cyklom (vymeni hodnotu, otestuje navrat atd)
2) ak je pred vyvolanim instrukcie xchg hodnota l->locked 0, aspon jedno CPU, ktore vyvolava tuto instrukciu nad zamkom 'uvidi' hodnotu 0; nastavi hodnotu 1, instrukcia xchg vrati hodnotu 0, takze test bude splneny a cyklus while(1) sa prerusi

spin lock:

!!! typ zamku, ktory neustale (v cykle) vytazuje testom hodnoty CPU !!!
zbytocne vytazuje CPU? preco neprepnut vykonavanie ineho procesu/vlakna, ktore nemusi cakat?
implementacia tohto typu zamkov je jednoduchsia ako zamkov s blokovanim, menej rezie
pravidla pre spravne pouzivanie zamkov typu spin:

1) drzanie zamku iba na veeeeelmi kratky cas!
2) v ziadnom pripade sa nikdy pri drzani zamku nevzdat CPU volanim yield!

'klasicky' blokujuci zamok:

cakajuce vlakna sa vzdavaju procesora a zaraduju sa do fronty
rezia je ovela vyssia ako pri type 'spin' (zaradovanie do fronty, planovac, prepnutie ctx)

zaverecne odporucania:

1) nikdy nezdielajte medzi vlakna/procesy nic, co nemusite
2) odladte aplikaciu pre beh na jednom jadre
3) identifikujte kriticke oblasti: neatomicke operacie, subehy (operacii a/alebo pamate)
4) vyvoj zacnite s velkymi zamkami; avsak iba na miestach, ktore vyzaduju vynutenie serializacie
5) jemnejsiu granularitu zamkov pouzite iba v krajnej nudzi (kvoli zvyseniu paralelneho vykonu)
6) pouzivajte automatizovane nastroje na hladanie subehov a uviaznuti