SU Media:

Student | Středník | ČWUT | AVC SH | Akropolis | iKariéra

Synchronizace paralelních procesů

<small>Z ČWUT</small>

Obsah

Časově závislé chyby, kritické sekce, vzájemné vyloučení

- časově závislé chyby
  - Dva nebo několik procesů používá (čte/zapisuje) společné sdílené prostředky (např. sdílená paměť, sdílení proměnné, sdílené soubory,…).
  - Výsledek výpočtu je závislý na přepínání kontextu jednotlivých procesů, které používají sdílené prostředky.
  - Velmi špatně se detekují (náhodný výskyt)!

- kritická sekce = část programu, kde procesy používají sdílené prostředky (např. sdílená paměť, sdílená proměnná, sdílený soubor, ...).
  - Sdružené kritické sekce = kritické sekce dvou (nebo více) procesů, které se týkají stejného sdíleného prostředku.

- vzájemné vyloučení
  - Procesům není dovoleno sdílet stejný prostředek ve stejném čase.
  - Procesy se nesmí nacházet ve sdružených sekcích současně.

Metody vzájemného vyloučení

zákaz přerušení (Disable Interrupt = DI)

- CPU je přidělováno postupně jednotlivým procesům za pomoci přerušení od časovače nebo jiného přerušení.
- Proces zakáže všechna přerušení před vstupem do kritické sekce a opět je povolí až po opuštění kritické sekce.
- Nevýhoda:
  - DI od jednoho uživatele blokuje i ostatní uživatele.
  - Ve víceprocesorovém systému, DI má efekt pouze na aktuálním CPU.
  - Zpomalí reakce na přerušení.
  - Problém se špatně napsanými programy (zablokují CPU)

- Užitečná technika uvnitř jádra OS (ale pouze na krátký čas)
- Není vhodná pro běžné uživatelské procesy!!!

povinné střídání

```
/* Process A */
while (TRUE) {
  while (turn != 0); /* wait */
  critical_section();
  turn=1;
  noncritical_section();
}

/* Process B */
while (TRUE) {
  while (turn != 1); /* wait */
  critical_section();
  turn=0;
  noncritical_section();
}
```

- Nevýhody
  - Jeden proces může zpomalit ostatní procesy.
  - Proces nemůže vstoupit do kritické sekce opakovaně (je porušen bod 3 z nutných podmínek ...).

Dekker-Petersonův algoritmus

```
#define FALSE 0
#define TRUE  1
#define N     2    /* number of processes */

int turn;          /* whose turn is it? */
int interested[N]; /* all values initially FALSE */

void enter_section (int process) /* process is 0 or 1 */
{
  int other;                     /* number of other processes */
  other = 1 - process;           /* the opposite of process */
  interested[process] = TRUE;    /* show that you are interested */
  turn = process;                /* set flag */
  while (turn == process && interested[other]); /* busy waiting */
}

void leave_section (int process) /* process: who is leaving */
{
  interested[process] = FALSE;   /* indicate departure from CS */
}
```

instrukce TSL

- Test and Set Lock (TSL) instrukce načte obsah slova z dané adresy v paměti do registru a nastaví obsah slova na nenulovou hodnotu.
- CPU provádějící TSL instrukci zamkne paměťovou sběrnici, aby znemožnilo ostatním CPU v přístupu do paměti dokud se instrukce TSL nedokončí.
- Korektní hardwarové řešení.
- TSL může být použita při synchronizaci v multiprocesorových systémech se sdílenou pamětí.

```
enter_section:
  TSL REGISTER, LOCK | copy LOCK to REGISTER and set LOCK to 1
  CMP REGISTER, #0   | was LOCK zero?
  JNE enter_section  | if it was non zero, LOCK was set, so loop
  RET                | return to caller, critical section entered

leave_section:
  MOVE LOCK, #0      | store a 0 in LOCK
  RET                | return to caller
```

operace sleep a wakeup

- Sleep()
  - Systémové volání, které zablokuje proces, který ho zavolal.
  - Zakáže alokování CPU pro tento proces a přesune ho do čekací fronty.

- Wakeup(proces)
  - Opačná operace, proces je uvolněn z fronty čekajících procesů a bude mu opět přidělováno CPU.

semafory

- Datový typ semafor obsahuje čítač a frontu čekajících procesů.
- Nabízí tři základní operace:
  - Init(): Čítač se nastaví na zadané číslo (většinou 1) a fronta se vyprázdní.
  - Down(): Pokud je čítač větší než nula, potom se sníží o jedničku. V opačném případě se volající proces zablokuje a uloží do fronty.
  - Up(): Pokud nějaké procesy čekají ve frontě, potom se první z nich probudí. V opačném případě se čítač zvětší o jedničku.
- Každá z těchto instrukcí je prováděna atomicky (tzn. že během jejího provádění nemůže být přerušena)

- Počáteční hodnota čítače semaforu určuje kolik procesů může sdílet nějaký prostředek současně.
- Z počáteční a aktuální hodnoty lze potom určit, kolik procesů je v daném okamžiku v kritické sekci.
- Binární semafor
  - Je často nazýván mutex (mutual exclusion).
  - Čítač semaforu je na začátku nastaven na jedničku.
  - Umožňuje jednoduše synchronizovat přístup do kritické sekce.

monitory

- Problém se semafory = Pokud například zapomenete použít operaci up() na konci kritické sekce, potom procesy čekající ve frontě budou čekat navždy (uváznutí).

- Řešení
  - Vyšší úroveň synchronizace se nazývá monitory.
  - Monitor je konstrukce vyšších programovacích jazyků, která nabízí stejné možnosti jako semafor.
  - Pouze jeden proces může být prováděn uvnitř monitoru v jednom okamžiku.
  - Monitory byly implementovány například v Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java, .NET,…

- Monitor
  - Množina procedur, proměnných a datových struktur, které jsou seskupeny dohromady ve speciálním modulu.
  - Pouze jeden proces může být aktivní v daném monitoru v jednom okamžiku.
  - Překladač (nikoliv programátor) se stará o vzájemné vyloučení uvnitř monitoru.

- Podmíněné proměnné
  - Předdefinovaná datový typ, který umožní pozastavit a opět spustit běžící proces.

- Operace
  - Wait(c): Pozastaví volající proces na podmíněné proměnné c.
  - Signal(c): Spustí některý z pozastavených procesu na podmíněné proměnné c.

zasílání zpráv

- Dvě základní operace:
  - send(destination,&message),
  - receive(source,&message).

- Synchronizace
  - Blokující send, blokující receive (rendezvous).
  - Neblokující send, blokující receive.
  - Neblokující send, neblokující receive + test příchozích zpráv.

- Adresování
  - Přímé: zpráva je uložena přímo do prostoru daného příjemce.
  - Nepřímé: zpráva je uložena dočasně do sdílené datové struktury (mailbox). To umožňuje lépe implementovat kolektivní komunikační algoritmy (one-to-one, one-to-many, many-to-many).

Klasické synchronizační úlohy a jejich řešení

producent a konzument

Jeden z klasických synchronizačních problémů, který demonstruje problémy paralelního programování.

- Producent produkuje nějaká data a vkládá je do sdílené paměti.
- Konzument vybírá data ze sdílené paměti.

Řešení s využitím sleep a wake-up

```
define N 100   /* number of slots in the buffer */
int count = 0; /* number of items in buffer */

void producer() {
  int item;
  while(TRUE) {
    item = produce_item();
    if (count == N)
      sleep(); /* if buffer is full, go to sleep */
    insert_item(item);
    count = count + 1;
    if (count == 1) wakeup(consumer); /* was buffer empty? */
  }
}

void consumer() {
  int item;
  while (TRUE) {
    if (count == 0) /* critical point for context switching */
      sleep(); /* if buffer is empty, go to sleep */
    remove_item(&item)
    count = count – 1;
    if (count == N - 1) wakeup(producer); /* was buffer full? */
    consume_item(&item);
  }
}
```

- Problém
  - Operace wakeup() je zavolána na proces, který není ještě uspán. Co se stane???
- Řešení
  - Wakeup waiting bit
  - Když je wakeup() zavolána na proces, který ještě nespí, tento bit je nastaven.
  - Při pokusu uspat proces, který má nastaven tento bit, proces se neuspí, ale pouze se resetuje daný bit.

Řešení s využitím semaforů

- Poznámka: Ochrana kritické sekce je požadována pouze v případě, že je více než-li jeden producent či konzument

```
#define N 100        /* number slots in buffer */
semaphore mutex = 1; /* guards critical section (CS)*/
semaphore empty = N; /* counts empty slots*/
semaphore full = 0;  /* counts full slots*/

void producer() {
  itemtype item;
  while (TRUE) {
    produce_item(&item);
    down(&empty);
    down(&mutex); /* enter CS*/
      enter_item(item);
    up(&mutex); /* leave CS*/
    up(&full);
  }
}

void consumer(void) {
  itemtype item;
  while (TRUE) {
```

```
    down(&full);
    down(&mutex); /* enter CS */
      remove_item(&item);
    up(&mutex); /* leave CS */
    up(&empty);
    consume_item(item);
  }
}
```

Řešení s využitím monitorů

```
procedure Producer;
begin
  while true do
  begin
    item := produce_item(item);
    Buffer.Insert(item);
  end
end;

procedure Consumer;
begin
  while true do
  begin
    Buffer.Remove(item);
    consume_item(item);
  end
end;

monitor Buffer
begin_monitor
  var Buff  : array[0..Max] of ItemType;
      count : integer;
      Full, Empty : condition;

  procedure Insert(E : ItemType);
  begin
    if count = N then Wait(Full);
    insert_item(E);
    count := count + 1;
    if count = 1 then Signal(Empty);
  end;

  procedure Remove(var E : ItemType);
  begin
    if count = 0 then Wait(Empty);
    remove_item(E);
    count := count - 1;
    if count = N - 1 then Signal(Full);
  end;

  count:=0;
end_monitor;
```

Řešení s pomocí zasílání zpráv

```
#define N 100 /* slots in shared buffer */

void producer(void) {
  int item;
  message m, e;
  while (TRUE) {
   item = produce_item();
   receive(consumer,&e); /* wait for an empty */
   build_message(&m,item); /* construct a message */
   send(consumer,&m); /* send to consumer */
  }
}

void consumer(void) {
  int item, i;
```

```
  message m, e;
  for (i = 0; i < N; i++) send(producer, &e); /* send N empties */
  while (TRUE) {
    receive(producer, &m); /* get message */
    item = extract_item(&m) /* extract item from message */
    send(producer,&e); /* send back empty reply */
    consume_item(item);
  }
}
```

večeřící filozofové

- Model procesů, které soutěží o výlučný přístup k omezenému počtu prostředků.
- N filozofů sedí kolem kulatého stolu a každý z nich buď přemýšlí nebo jí. K jídlu potřebuje současně levou a pravou vidličku.

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)
enum stat(thinking, hungry, eating);
enum stat state[N];
semaphore mutex=1;
semaphore s[N]; # initially set to 0

void philosopher(int i) {
  while (TRUE) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}

void take_forks(int i) {
  down(&mutex);
    state[i] = hungry;
    test(i);
  up(&mutex);
  down(&s[i]);
}

void put_forks(int i) {
  down(&mutex);
    state[i] = thinking;
    test(LEFT);
    test(RIGHT);
  up(&mutex);
}

void test(int i) {
  if (state[i] == hungry && state[LEFT] != eating && state[RIGHT] != eating) {
    state[i] = eating;
    up(&s[i]);
  }
}
```

http://www.doc.ic.ac.uk/~jnm/book/book_applets/Diners.html

čtenáři a písaři

- Model procesů, které přistupují do společné databáze.
- Více čtenářů může číst současně data pokud žádný písař nemodifikuje data v databázi.
- Pouze jeden písař může modifikovat data v databázi v jednom okamžiku.

```
int rc = 0; /* readers counter */
semaphore mutex = 1;
semaphore db = 1; /* access to database */

void reader(void) {
```

```
  while(TRUE){
    down(&mutex);
      rc = rc + 1;
      if (rc == 1) down(&db);
    up(&mutex);
    read_data_base(); /* crit. section */
    down(&mutex);
      rc = rc - 1;
      if (rc == 0) up(&db);
    up(&mutex);
    use_data_read(); /* non crit. sect.*/
  }
}

void writer() {
  while(TRUE) {
    think_up_data();
    down(&db);
      write_data_data();
    up(&db);
  }
}
```

spící holič

V holičství je N holičů (barber), N holících křesel (barber chair) a M čekacích křesel (waiting chair) pro zákazníky. • Pokud není žádný zákazník v holičství, holič sedne do holícího křesla a usne. • Pokud přijde zákazník, potom

1. pokud nějaký holič spí, tak ho probudí a nechá se ostříhat,
2. jinak si sedne čekacího křesla a čeká (pokud nějaké je volné),
3. jinak opustí holičství.

```
#define CHAIRS 5
int waiting = 0; /* customers are waiting (not being cut) */
semaphore mutex = 1; /* for mutual exclusion */
semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0; /* # of barbers waiting for customers */

void Customer() {
  down(&mutex);
  if (waiting < CHAIRS) {
    waiting = waiting + 1;
    up(&customes);
    up(&mutex);
    down(&barbers);
    get_haircut();
  } else {
    up(&mutex);
    leave_shop_without_haircut();
  }
}


void Barber() {
  while (TRUE) {
    down(&customers);
    down(&mutex)
    waiting = waiting - 1;
    up(&barbers);
    up(&mutex);
    cut_hair();
  }
}
```

Stránka byla naposledy editována v 12:02, 13. 6. 2006.
