Mekanismer

(implementation)

Repetition

- Semafor
  - Räknar tillgängliga resurser
  - Initieras med startvärde
  - Vid förbrukning: väntar tills resurs finns
  - Användning: invänta händelse
- Lås
  - Markerar att en variabel/datastruktur är upptagen
  - Har ägare och felhantering
  - Vid låsning: väntar tills variabeln/datastrukturen är ledig
  - Användning: skapa mutual exclusion
- Condition
  - Placerar låst tråd i vänteläge
  - Har inget minne
  - Användning: invänta ändring av låst variabel/datastruktur

Hur implementeras de?

- Skall förhinda alla race-conditions
  - skall garantera deterministikt resultat
- Är i sig själva kritiska sektioner
  - rekursiv catch 22
  - för att garantera enskild access till en resurs krävs enskild access till (en annan) resurs osv...
- Skall fungera effektivt
  - busy-wait förbjudet
  - nyttjar inga resurser vid väntan, placeras på väntekö
  - progress
  - bounded waiting
- Skall fungera på både uniprocessor och multiprocessor

Kritisk sektion – 4 delar

- Ingång (entry section)
  - Hur implementeras ingången till en kritisk sektion?
- Kritisk sektion (critical section)
  - Kod som kräver mutual exclusion
- Utgång (exit section)
  - Hur implementeras utgången
- Återstod (remainder section)
  - Representerar normal, icke-kritisk kod

Kritisk sektion - exempel

Några trådar kör följande kod.

```
forever
{
    lock()                          // ingång
    global = global + 1             // kritiskt
    unlock()                        // utgång
    ++local                         // återstod
}
```

Tre krav ställs

- Mutual Exclusion
  - When a given process is in its critical section other processes cannot be in their critical sections.
- Progress
  - The decision as to who can enter the critical section must not be postponed indefinitely. Also, when no process is in its critical section and several processes are requesting to enter their critical section, who should decide who will enter the critical section? Answer: Only those processes which are not in their remainder section.
- Bounded Waiting
  - If process P1 makes a request to enter its critical section, the request might not be granted right away. Instead other processes may be allowed to enter their CSs. Process P1 must then wait. If there is a limit to the number of times those other processes may be granted access to their CSs before P1's request is granted then it is a bounded wait. I.e. waiting is not indefinite.

Live-lock

Något de flesta någon gång fått en känsla för i verkliga livet....

- Tänk dig två personer som skall mötas
- Båda stiger åt sidan åt samma håll för att låta den andra passera
- Båda stiger åt sidan åt andra hållet
- Förprogrammerade processer skulle fortsätta enligt detta mönster i evighet...
- Människor skrattar förläget till varandra, ber om ursäkt, och löser situationen enligt nytt protokoll...

Låset – eget försök

- Implementera ett lås (”upptagetskylt”) som skyddar en kritisk variabel/datastruktur
- Vid låsning
  - tråd skall vänta tills ledigt
- Vid upplåsning
  - väntande tråd(ar) vaknar
- Krav
  - Mutual exclusion upprätthålls
  - Inga krav på effektivitet

Några försök

- Använd en ”upptaget”-flagga
- Peterson’s lösning (för två trådar)
- Förhindra trådbyte geom att hindra avbrott
- Kombinera!
- Placera på väntekö
- Nyttja speciella hårdvaruinstruktioner

Utför steg för steg och diskutera.

Försök med flagga (FEL!)

```
bool busy = false; /* init */

while ( busy )
   ;
busy = true;

/*
 * godtyckligt lång kritisk sektion här
 */

busy = false;
```

Peterson's

```
i_want_in[ME(id)] = true;
whose_turn = OTHER(id);

while ( i_want_in[OTHER(id)] &&
        whose_turn == OTHER(id) )
  ; /* busy wait */
/* insert critical code here */
i_want_in[ME(id)] = false;
```

Analys av Peterson's

- Mutex:
  - P_i får tillträde omm det är hens tur eller den andre inte vill in
  - Om P_j också vill in så är antingen P_i eller P_j i loopen
- Progress (prevent livelock)
  - Om P_i vill in kommer den komma in direkt eller ...
- Bounded waiting
  - ... efter att P_j smiter före högst en gång

Avbrott? (DÅLIGT)

```
old_level = interrupt_disable()

/*
 * kritisk sektion här
 */

interrupt_enable(old_level)
```

- Riskfyllt
- Missar avbrott eller fördröjer avbrottshantering
- Ej tillgängligt för usermodetrådar
- Fungerar inte på multiprocessorer
- Stängs avbrott av måste kritiska sektionen vara mycket kort

Synkronisera avbrottshanterare

- Vid avbrott (interrupt) från hårdvara, t.ex. timer, så exekverar operativsystemet en funktion för att hantera avbrottet.
- Vad händer om den funktionen använder en delad resurs?
- Lås fungerar inte: Avbrottet kan ske när tråden som avbryts håller låset. Då försöker avbrottshanteraren (som lånar av trådens exekveringstid) låsa igen. Men det är förbjudet eftersom låset har felhantering som hindrar att en tråd låser när den redan håller låset.
- Binär semafor fungerar inte: Avbrottet kan ske när tråden som avbryts räknat ned semaforen. Då försöker avbrottshanteraren (som lånar av trådens exekveringstid) räkna ned igen och kommer att vänta på sema_up. Men det kommer inte att hända, för tråden exekverar ju avbrottshanteraren och kommer inte fortsätt med sin kod förrän avbrottshanteraren är klar.
- Avbrott *måste* stängas av för att synkronisera kod mellan trådar och funktioner som hanterar avbrott. Det är enda tillfället avbrott bör manipuleras.

Kombinera (FEL!)

```
bool busy = false; /* init */

old_level = interrupt_disable();
while ( busy )
   ;
busy = true;
interrupt_enable(old_level);

/* godtyckligt lång kritisk sektion här */

busy = false;
```

Väntekö (Pintos)

```
bool busy = false; /* init */
init_wait_queue(); /* init */

old_level = interrupt_disable();
while ( busy )
{
   place_me_on_wait_queue();
   switch_to_other_ready_thread();
}
busy = true;
interrupt_enable(old_level);

/* godtyckligt lång kritisk sektion här */

old_level = interrupt_disable();
busy = false;
wake_other_from_wait_queue_and_place_on_ready_queue();
interrupt_enable(old_level);
```

Generellt trådbyte

```
/* utförs med avbrott avstängt */
old_level = interrupt_disable();

place_me_on_ready_queue();
switch_to_other_ready_thread();

/* sätter på avbrott så fort
   tråden får fortsätta igen */
interrupt_enable(old_level);
```

Test and set

En assemblerinstruktion ekvivalent med:

```
bool test_and_set(bool* flag)
{
    bool save = *flag;
    *flag = true;
    return save;
}
```

Atomic swap

En assemblerinstruktion ekvivalent med:

```
void atomic_swap(int* a, int* b)
{
    int save = *a;
    *a = *b;
    *b = save;
}
```

Test and set igen

Kan implementeras race-free med atomic_swap:

```
bool test_and_set(bool* flag)
{
    bool set = true;
    atomic_swap(&set, flag);
    return set;
}
```

Lås med test and set

```
bool busy = false; /* init */

while (test_and_set(&busy))
   ;

/* kort kritisk sektion här */

busy = false;
```

Compare and swap

```
int compare_and_swap(
   int* mem,
   int  compare_to,
   int  replacement)
{
   int oldmem = *mem;
   if (oldmem == compare_to)
      *mem = replacement;
   return oldmem;
}
```

Lås med compare and swap

```
bool busy = false; /* init */

while (compare_and_swap(&busy,
                        false, true))
  ; /* noop */

/* kort kritisk sektion här */

busy = false;
```

Flera implementationer

Finns på wikipedia för den intresserade.

- Delad resurs – Två trådar
  - Dekker's algorithm
  - Peterson's algorithm
- Delad resurs – Många trådar
  - Eisenberg McGuire algorithm
  - Lamport’s bakery algorithm

Av speciellt intresse kan vara varianter för att ordna mutual exclusion i JavaScript med AJAX (Asynchronous Javascript And XML). Normala JavaScript kör typiskt sett i webbläsarens UI-tråd och saknar(?) inbyggt stöd för både trådar och mutual exclusion (såsom lås och semaforer). AJAX callbacks är dock asynkrona(?) och kan behöva synkronisering om de uppdaterar globala data. (Mer utredning av JS-expert krävs.)