# TDIU16 – Mekanismer

## Hur fungerar en semafor?

Filip Strömbäck, Klas Arvidsson

LINKÖPING UNIVERSITY

# Planering

| Vecka | Fö/Se | Lab |
|---|---|---|
| 13 | Fö: C + Syscall (påsk) | C[1], halt, exit, console |
| 14 | - | console, filhantering |
| 15 | Fö: Sema, Se: Process | main, första processen |
| 16 | Fö: Lås, cond. | exec, exit |
| 17 | Fö: Implementation | exec, exit, wait |
| 18 | Fö: Deadlock | Synkronisering |
| 19 | - | Synkronisering, accesskontroll |
| 20 | Se: Deadlock + tenta | Synkronisering, accesskontroll |
| 21 | - | Synkronisering, accesskontroll |

[1] lämpligt att demonstrera första passet

# Repetition

Semafor

- Räknar tillgängliga resurser
- Väntar tills resurs finns
- Användning: invänta händelse

Lås

- Markerar om en variabel/datastruktur används
- Väntar till kritisk sektion är ledig
- Användning: skapa mutual exclusion

Condition

- Placerar låst tråd i vänteläge
- Väntar tills någon anropar `signal` eller `broadcast`
- Användning: invänta händelse

# Hur är de implementerade?

Vi kommer huvudsakligen titta på lås idag.

- Samma idéer kan dock användas för att implementera andra primitiver, exempelvis semaforer.
- Samma typer av resonemang kan användas för att implementera synkronisering över nätverk (vi har dock andra byggstenar och antaganden då, se ex TDDD25).

# Antag följande

```
struct lock l;

void foo() {
  lock(&l);

  /**
   * Godtycklig kritisk sektion.
   */

  unlock(&l);
}
```

# Hur svårt kan det vara? (FEL)

```
struct lock { bool busy = false; };

void lock(struct lock *l) {
  while (l->busy)
    ; /* Vänta */
  l->busy = true;
}

void unlock(struct lock *l) {
  l->busy = false;
}
```

# Hur svårt kan det vara? (OK?)

```
struct lock {
  bool busy = false;
  struct lock lock;
  struct condition cond;
};
void lock(struct lock *l) {
  lock_acquire(&l->lock);
  while (l->busy)
    cond_wait(&l->cond, &l->lock);
  l->busy = true;
  lock_release(&l->lock);
}
```

# Hur svårt kan det vara? (OK?)

```
void unlock(struct lock *l) {
  lock_acquire(&l->lock);
  l->busy = false;
  cond_signal(&l->cond, &l->lock);
  lock_release(&l->lock);
}
```

Så, synkroniserat och klart!

# Hur svårt kan det vara? (OK?)

```
void unlock(struct lock *l) {
  lock_acquire(&l->lock);
  l->busy = false;
  cond_signal(&l->cond, &l->lock);
  lock_release(&l->lock);
}
```

Så, synkroniserat och klart!

Vänta nu... Det är ju vi som implementerar `lock`...

# Problem

- Ska förhindra *race conditions*
- Ska garantera deterministiskt resultat
- Innehåller någon typ av variabel/resurs, åtminstone *låst*/*upplåst*

  $\Rightarrow$ vi behöver synkronisering
- Vi kan dock inte använda lås/semaforer/etc.

# Vad kräver vi?

- Ska förhindra *race conditions*
- Ska vänta effektivt
  - $\Rightarrow$ ingen *busy-wait*
  - $\Rightarrow$ placera på väntekö
- Ska fungera oavsett hur många CPU vi har

# Formella krav

1. Mutual Exclusion
   - När en tråd är inuti en kritisk sektion kan inte andra trådar befinna sig i samma kritiska sektion.
2. Progress
   - Om en eller flera trådar begär tillgång till en tom kritisk sektion ska trådarna alltid besluta vilken som får tillgång till den kritiska sektionen inom ändlig tid.
3. Bounded Waiting
   - Om en tråd begär tillgång till en kritisk sektion och behöver vänta ska det finnas en gräns på antalet gånger som andra trådar får ”gå före” in i den kritiska sektionen. D.v.s. tråden ska få tillgång förr eller senare.

# 1: Ingen mutual exclusion

Om inte villkor 1: *mutual exclusion* är det som om vi inte hade synkroniserat alls, alltså väldigt dåligt!

# 2: Ingen progress

Utan villkor 2: *progress* kan *live-lock* uppstå.

Tänk dig två personer som går rakt emot varandra

- Båda stiger åt sidan åt samma håll för att låta den andra passera
- Båda stiger åt sidan åt andra hållet
- ...

Program skulle fortsätta i evighet, människor kan lösa problemet på annat sätt.

# 3: Ingen bounded waiting

Utan villkor 3: *bounded waiting* kan *starvation* uppstå.

*Starvation* händer då andra trådar alltid får gå före i väntekön, så att någon tråd kanske behöver vänta för evigt.

Kan leda till att en del av systemet aldrig får en chans att köra, och en del av systemet fryser.

# Idé

Antag att vi har två trådar.

I stället för att ha en delad variabel:

- En variabel för varje tråd
- Ytterligare en variabel som berättar vems tur det är

- Släpps in om bara jag vill in, eller om det är min tur

# Implementation

```
struct lock {
  // vilka trådar vill in i kritisk sektion?
  bool want_in[2] = { false, false };

  // vems tur är det att gå först?
  int turn = 0;
};

// tråd-id för nuvarande tråd
int ME();
// tråd-id för andra tråden
int OTHER();
```

# Implementation

```
void lock(struct lock *l) {
  l->want_in[ME()] = true;
  l->turn = OTHER();

  while (l->want_in[OTHER()]
      && l->turn == OTHER())
    ; /* Busy wait */
}

void unlock(struct lock *l) {
  l->want_in[ME()] = false;
}
```

# Fördelar

- Uppfyller villkoren för kritisk sektion
  - Mutual exclusion
  - Progress
  - Bounded waiting
- Kräver inget speciellt hårdvarustöd
- Fungerar för fler CPU (beroende på minnesmodell)

# Nackdelar

- Fungerar endast för två trådar (går att utöka, men blir snabbt ohållbart)
- Busy wait i nuläget (går att lösa)
- Antar att läsningar och skrivningar till minnet beter sig som i koden
  - Stämmer inte på vissa arkitekturer
  - Stämmer inte då kompilatorn optimerar koden

# Idé

OS använder interrupt för att göra trådbyten

- Om vi stänger av interrupt så kan inga trådbyten ske

$\Rightarrow$ ingen preemption

$\Rightarrow$ mutual exclusion!

# Implementation (DÅLIGT)

```
struct lock {
  int old_level;
};

void lock(struct lock *l) {
  l->old_level = interrupt_disable();
}

void unlock(struct lock *l) {
  interrupt_enable(l->old_level);
}
```

# Nackdelar

- Fördröjd hantering av avbrott
- För att undvika problem måste den kritiska sektionen vara mycket kort
- Ej tillgänglig från usermode
- Fungerar inte på multiprocessorer
- Uppfyller inte nödvändigtvis *bounded waiting*
- (var försiktig med optimeringar)

# Fördelar

- Ger mutual exclusion på system med en CPU
- Måste användas för att synkronisera avbrottshanterare
  - Lås fungerar inte: avbrott kan avbryta den kritiska sektionen och försöka låsa ifrån samma tråd igen ⇒ låsets felhantering ger `PANIC`
  - Semaforer fungerar inte: avbrott kan avbryta den kritiska sektionen och anropa `sema_down` ifrån samma tråd igen ⇒ deadlock
  - Fundera noga på vilken eller vilka CPU som hanterar avbrott då multiprocessorsystem används!

# Kombinera med busy flag (FEL)

```
struct lock { bool busy = false; };

void lock(struct lock *l) {
  int old = interrupt_disable();
  while (busy)
    ;
  busy = true;
  interrupt_enable(old);
}

void unlock(struct lock *l) {
  l->busy = false;
}
```

# Trådbyte (Pintos)

```
// Trådbyte görs med avbrott avstängda
int old_level = interrupt_disable();

place_me_on_ready_queue();
switch_to_other_thread();

// Sätt på avbrott så fort tråden får fortsätta
interrupt_enable(old_level);
```

# Väntekö (Pintos)

```
void lock(struct lock *l) {
  int old = interrupt_disable();
  while (busy) {
    add_to_wait_queue(&l);
    thread_block(); // Byt tråd
  }
  busy = true;
  interrupt_enable(old);
}
```

# Väntekö (Pintos)

```
void unlock(struct lock *l) {
  int old = interrupt_disable();
  l->busy = false;
  struct thread *t = get_from_queue(&l);
  thread_unblock(t);
  interrupt_enable(old);
}
```

# Idé

De flesta CPU:er tillhandahåller *atomiska operationer*.
Kan vi använda dem på något sätt?

- Atomiska operationer läser och/eller modifierar minnet på något sätt
- Hårdvaran garanterar att atomiska operationer inte avbryts av varandra
- Alltså: atomiska operationer ser inte halvklara effekter av andra atomiska operationer
- På X86 är även läsning/skrivning till minnet också atomärt (givet korrekt *alignment*, men se upp för optimeringar av kompilatorn)

# Test and set

Instruktion ekvivalent med:

```
bool test_and_set(bool *flag) {
  lock_acquire(&global_lock);
  bool save = *flag;
  *flag = true;
  lock_release(&global_lock);
  return save;
}
```

# Lås med test and set

```
struct lock { bool busy = false; };

void lock(struct lock *l) {
  while (test_and_set(&l->busy))
    ;
}

void unlock(struct lock *l) {
  l->busy = false;
}
```

# Atomic swap

Instruktion ekvivalent med:

```
int atomic_swap(int *mem, int replace) {
  lock_acquire(&global_lock);
  int tmp = *mem;
  *mem = replace;
  lock_release(&global_lock);
  return tmp;
}
```

I C++: `exchange`

# Lås med atomic swap

```
struct lock { int busy = 0; };

void lock(struct lock *l) {
  while (atomic_swap(&l->busy, 1) == 1)
    ;
}

void unlock(struct lock *l) {
  l->busy = 0;
}
```

# Compare and swap (CAS)

Instruktion ekvivalent med:

```
int compare_and_swap(int *mem,
                     int compare,
                     int replace) {
  lock_acquire(&global_lock);
  int old = *mem;
  if (old == compare)
    *mem = replace;
  lock_release(&global_lock);
  return old;
}
```

I C++: `compare_exchange`

# Test and set igen

```
int test_and_set(int *mem) {
  return compare_and_swap(mem, 0, 1);
}

int test_and_set(int *mem) {
  return atomic_swap(mem, 1);
}
```

# Lås med compare and swap

```
struct lock { int busy = 0; };

void lock(struct lock *l) {
  while (compare_and_swap(&l->busy, 0, 1))
    ;
}

void unlock(struct lock *l) {
  l->busy = 0;
}
```

# Mer komplexa operationer

```
void decrement_if_possible(int *value, int dec) {
  do {
    // 1: Read data.
    int orig = atomic_read(value);
    // 2: Modify data.
    int write = orig;
    if (write >= dec)
      write -= dec;
    // 3: Commit changes.
  } while (orig !=
      compare_and_swap(value, orig, write));
}
```

# Andra operationer

X86 har också några ytterligare atomiska operationer som kan vara användbara, exempelvis:

```
atomic_add          x += y
atomic_sub          x -= y
atomic_increment    x++
atomic_decrement    x--
```

Se `https://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Atomic-Builtins.html` för mer info

# Är atomiska operationer snabbare än lås?

Vi kan enkelt testa i `sum`!

Vilken lösning är bäst?

Är de atomiska operationerna tillräckligt kraftfulla för att synkronisera bankproblemet?

Filip Strömbäck, Klas Arvidsson

www.liu.se