Hoofstuk 7 - Sinchronisasie van Prosesse

- Probleem: Prosesse deel data, maar is dit konsistent?
- Voorbeeld: *Producer-Consumer*
- Vervlegging van masjienkode in gelyklopende stelsels
- Naelkondisie (*race condition*)

Producer:

```
while (1) {
  while (counter == BUFFER_SIZE);
  buffer[in] := produce_item();
  in = (in+1) % BUFFER_SIZE;
  counter = counter+1;
}
```

Consumer:

```
while (1) {
  while (counter == 0);
  consume_item(buffer[out]);
  out = (out+1) % BUFFER_SIZE;
  counter = counter-1;
}
```


Hoofstuk 7 - Kritieseseksie

- Stelsel met n prosesse: $\{P_0, P_1, ..., P_{n-1}\}$
- Kenmerk van die stelsel: Slegs **een** proses mag in sy kritieseseksie wees
- Onderlinguitsluitend (*mutual exclusion*)
- Drie vereistes moet bevredig word:
  1. Onderlinguitsluitend
  2. Vordering
  3. Begrensde wagtyd

Proses i

```
do {
  while (turn != i);
  /* Critical Section */
  turn = j
  /* Remainder Section */
} while (1);
```

Proses j

```
do
  while (turn != j);
  /* Critical Section */
  turn = i
  /* Remainder Section */
} while (1);
```

Hoofstuk 7 - Sinchronisasie Hardeware

- Kan die kritieseseksie probleem opgelos word deur onderbrekings af te skakel?
- Wat gebeur as die stelsel veelvoudige verwerkers bevat?
- Wat is die implikasies vir tyddeelstelsels?
- *Test-and-Set* en *Swap* instruksies is atomies.


```
do {
  waiting[i] = true;
  key = true;
  while (waiting[i] && key)
    key = test_and_set(lock);
  waiting[i] = false;
  /* Critical Section */
  j := (j+1) % n;
  while ((j != i) && !waiting[j])
    j = (j+1) % n;
  if (j == i) lock = false
  else waiting[j] = false;
  /* Remainder Section */
} while (1);
```


Hoofstuk 7 - Semafore

- Definisie
- Atomiese operasies: *Signal* (V) en *Wait* (P)
- 'n Enkele gedeelde semafoor kan gebruik word om n prosesse te sinchroniseer
- Semafore kan sekere uitvoervolgordes afdwing
- Implementasie: *Spinlocks*, konteksverandering en toue
- Vergrendeling (*deadlock*) en verhongering (*starvation*)

Hoofstuk 7 - Voorbeeld van Semafore

Elke proses in die stelsel het die volgende struktuur:

```
do {
  wait(mutex);
  /* Critical Section */
  signal(mutex);
  /* Remainder Section */
} while (1);
```

Veronderstel daar is n prosesse: $\{P_0, P_1, ..., P_{n-1}\}$ en 'n semafoor, `mutex`. Die volgende scenario is dan moontlik:

1. P_0 voer `Wait(mutex)` uit, dus `mutex = 0`
2. P_0 word onderbreek. $P_1..P_{n-1}$ wag omdat `mutex = 0`
3. P_0 betree sy kritieseseksie terwyl die ander prosesse nog steeds wag
4. P_0 voer `Signal(mutex)` uit, dus `mutex = 1`
5. 'n Ander proses kan nou sy kritieseseksie binnegaan

Hoofstuk 7 - Gevalle Studies

- *Readers-Writers*
- *Dining Philosophers*


```
semaphore mutex, wrt;
int readcount;
```

Writer:

```
wait(wrt);
/* Write something */
signal(wrt);
```

Reader:

```
wait(mutex);
readcount = readcount+1;
if (readcount == 1) wait(wrt);
signal(mutex);
/* Read something */
wait(mutex);
readcount = readcount-1;
if (readcount == 0) signal(wrt);
signal(mutex);
```


Hoofstuk 7 - Kritiese Area

- Probleme kom steeds voor indien semafore verkeerd gebruik word
- Kritiese areas is 'n taal konstruk en poog om die aantal foute te verminder
- Gedeelde veranderlikes eksplisiet verklaar en slegs binne 'n kritiese area toegangklik, bv.

```
  T shared v;
...
region v when B {
  S
}
```

- Implikasies vir vertalers

- Taal konstruk wat beide data en prosedures bevat om gedeelde bron te bestuur (Dijkstra, 1971)

- Prosedures en veranderlikes slegs toeganglik binne die monitor

- Sinchronisasie d.m.v. kondisie (*condition*) veranderlikes

- Operasies *Signal* en *Wait* word herdefinieer vir kondisie veranderlikes.