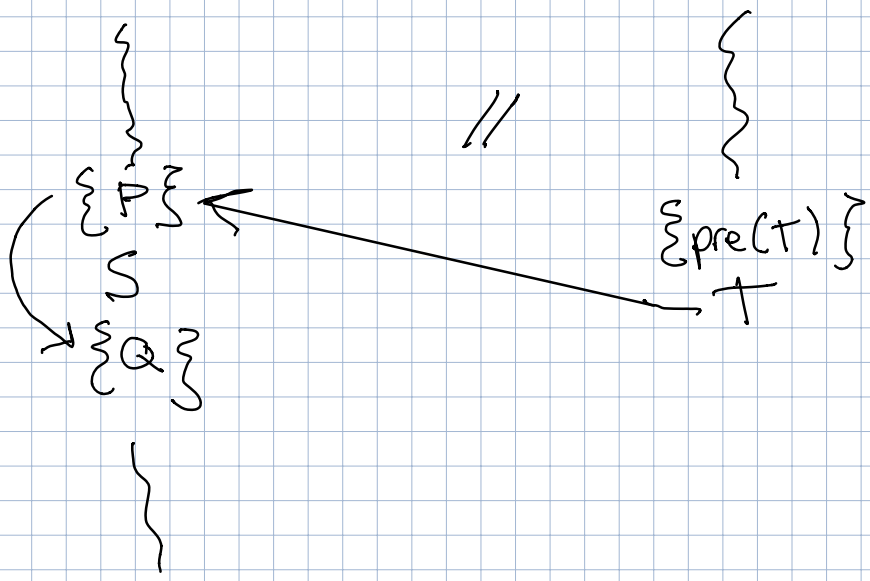
Concurrency (9/17/2019 CS4410 F Schneider)

process, thread, core, processor....
- finite progress
- shared memory

cooperation → communication + synchronization

synchronization
- mutual exclusion
  - "critical section problem"
- await B
  - spinning / busy-wait

classic problems



interference-freedom

Interlock Instructions

Hw instruction that reads & updates shared memory as part of a single atomic action.

Eg "test and set"

TS(S, l): ⟨ l := S ; S := true ⟩

indivisible hw op

S: typically shared memory

l: typically not shared (eg a reg)

Mutual Exclusion using TS

var s: ~~boolean~~ initial (false)
var l_1 l_2 ... l_n: ~~boolean~~ initial (true)

I: At most one of $s, l_1, l_2 \ldots l_n$ is false

$$\wedge \left(\bigwedge_i in(CS_i) \Rightarrow \neg l_i \right)$$

Proof that I suffices to establish mutual exclusion (assume I is invariant)

$$in(CS_1) \wedge in(CS_2) \wedge I$$
$$\Rightarrow$$
$$\neg l_1 \wedge \neg l_2 \wedge \text{At most one... is false}$$
$$\Rightarrow$$
false!

I: At most one of $s\ l_1, l_2 \ldots l_n$ is false

$$\wedge \left(\bigwedge_i in(CS_i) \Rightarrow \neg l_i \right)$$

---

entry$_i$: $\{I \wedge l_i\}$
while l_i do $\{I \wedge l_i\}\ TS(S, l_i)\ \{I\}$
$\{I \wedge \neg l_i\}$

exit$_i$ $\{I \wedge \neg l_i\}\ l_i := true$
$\{s = l_1 = l_2 = \ldots\ l_n = true \wedge I\}\ s := false\ \{I\}$

Other interlock instructions

Swap(a, b): $\langle a, b := b, a \rangle$

$CS(t, x, o, n)$: $\langle t := (o = x)$
if t then $x := n$
else $o := x \rangle$

Synchronization Primitives

... prevent wasteful busy-wait (spinning)

await B

⇓

while ¬B do skip end

⇓

while ¬B do yield end

Where is state B stored?

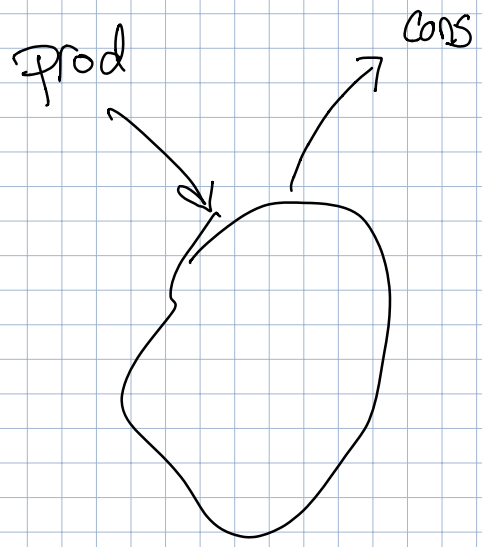
- If stored in OS then sys call can change its value & OS will "know" when to awaken a waiting process

Result: OS implemented synchronization primitives

Producer/Consumer Problem

What kinds of synchronization is needed?

Prod: do forever
Create item
await free slot
entry
insert item
exit
end



//

Cons: do forever
await full slot
entry
retrieve item
exit
use item
end

Requirements for synchronization primitives

- General purpose:
  - mutual exclusion
  - condition synchronization (await B)
- Simple to use

Many primitives have been provided in various systems

Semaphores (Dijkstra 1962)

sem: non negative integer

P(sem): ⟨await sem > 0 then sem := sem − 1⟩

V(sem) ⟨sem := sem + 1⟩

Many notate ⟨S⟩ to signify indivisible operation...

Some history

- Introduced by Dijkstra in THE system circa 1962

P stands for **procure**

Proberen "to test" or "to try"

Passeren "passing"

but Dijk later said

Prolaag short for "probeer te verlagen"

try to reduce

V stands for **vacate**

Verhogen "increase"

Vrijgave "release"

CS4410 terms

Semaphores admit simple solution to mutex

var mutex: semaphore init(1)

P1: do forever
P(mutex)
CS_1
V(mutex)
NCS_1
end

//

P2: do forever
P(mutex)
CS_2
V(mutex)
NCS_2
end

Why does above protocol implement mutual exclusion?

what we desire

$I \Rightarrow$ at most 1 process exec in CS

I: $1 - mutex = \#processes\ in\ CS$
$\wedge \quad 0 \le mutex \le 1$

proposal

Adds 1 to # of processes in CS (RHS)
Subt 1 from mutex
(= adds 1 to LHS)

$\{I\}$

$P(mutex)$: await $mutex > 0$ then $mutex := Mutex - 1$

$\{I \wedge mutex = 0\}$

CS_1

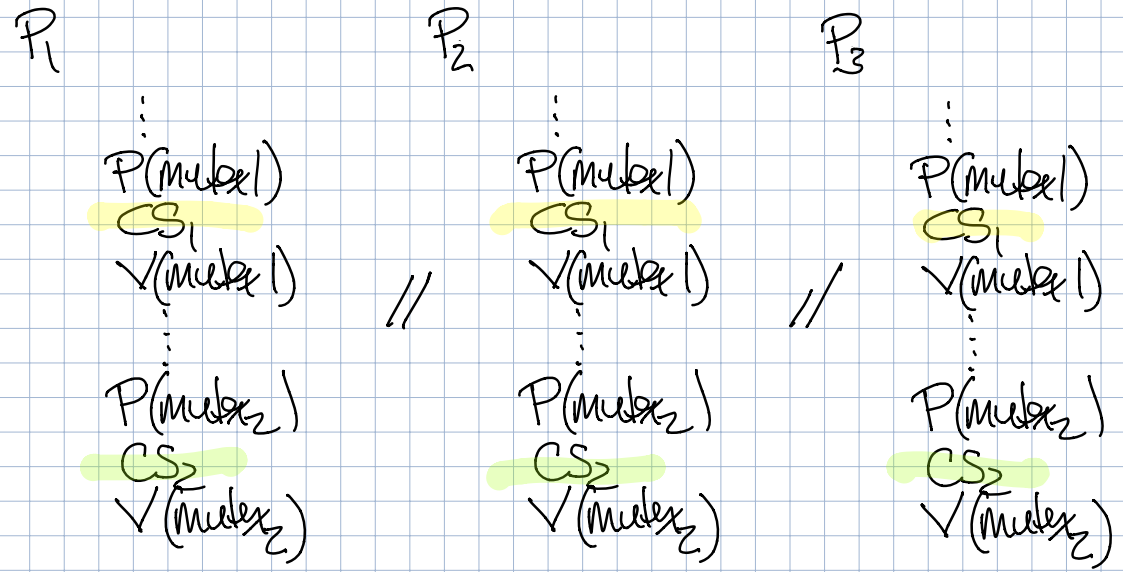
$\{I \wedge mutex = 0\}$

Adds 1 to mutex
(= subt 1 from LHS)
Subt 1 from # processes in CS
(RHS)

$V(mutex)$: $mutex := Mutex + 1$

$\{I\}$

Semaphores support selective mutual exclusion.
(only some subset of critical sections exclude)



Semaphores also support
k-exclusion.
= allow up to k critical sections to execute concurrently

Bounded Buffer Problem

var a: integer number items added
r: integer number items removed

Require: $0 \le a - r \le N$ ← size of buffer
↑ # of items stored

Prod:

await $a - r < N$ ← ensures space avail to add item

add item:

$a := a + 1$

//

Cons:

await $0 < a - r$ ← ensures items avail to remove

remove item

$r := r + 1$

await → mutex

① Mutual exclusion to protect bounded buffer
② Increments do not cause interference

var a: integer number items added
r: integer number items removed

var mutex: semaphore init (1)

I: $0 \le a - r \le N$

Prod:

await $a-r < N$ $\{a-r<N\}$
P(mutex)
add item:
$a := a+1$
V(mutex)

//

Cons:

await $0 < a-r$ $\{0<a-r\}$
P(mutex)
remove item
$r := r+1$
V(mutex)

await → mutex

Use slots to encode $N-(a-r)$
items to encode $a-r$

var a: integer number items added
r: integer number items removed

var mutex: semaphore init(1)
var slots: semaphore init(N)
items: semaphore init(0)

I: $0 \le a-r \le N \wedge slots = N-(a-r) \wedge items = a-r$

Prod:

{I}
P(slots): await slots > 0 then slots := slots - 1
{... $\wedge$ slots - 1 = N - (a+1-r) ...}
{... $\wedge$ slots = N - (a+1-r) ...}
await a - r < N
P(mutex)
add item:
a := a + 1
{... $\wedge$ slots = N - (a-r) ...}
V(mutex)
{I}

implies

...

Note: $slots > 0 \wedge slots - 1 = N - (a+1-r)$
$\Rightarrow$ $slots > 0 \wedge slots = N - (a-r)$
$\Rightarrow$ $a - r < N$

So P(slots) has same effect as
await $a - r < N$

var a: integer number items added
var r: integer number items removed

var mutex: semaphore init(1)
var slots: semaphore init(N)
var items: semaphore init(0)

I: $0 \le a - r \le N \wedge slots = N - (a - r) \wedge items = a - r$

// Cons:

{I}
P(items): await items > 0 then items := items - 1
await $0 < a - r$
P(mutex)
remove item
$r := r + 1$
V(mutex)
V(slots)

* does not imply!

{... $slots = N - (a - (r+1))$ ...}
{I}
{I}

* To fix: need ____ to be "slots + 1"
Soln: Add V(slots)

Similar analysis with respect to

I ... $\wedge$ items = a - r ...

justifies V(items) at end

```
prod: process.

    P(slots)
    P(mutex)
    add item
    a := a+1
    V(mutex)
    V(items)
end
```

Final Solution (multiple producers / multiple consumers)

```
var mutex semaphore init(1)
    slots semaphore init(N)
    items semaphore init(0)

prod: process
        P(slots)
        P(mutex)
        bb[a mod N] := item
        a := a+1
        V(mutex)
        V(items)
      end

cons: process
        :
        P(items)
        P(mutex)
        val := bb[r mod N]
        r := r+1
        V(mutex)
        V(slots)
      end
```

Special Case: 1 producer & 1 consumer.

Does producer ever insert into bb[a mod N] while consumer reads bb[r mod N] when

$$a \bmod N = r \bmod N$$

If no, then mutex not needed!

{ Inv: slots = N − (a − r)
items = a − r
$0 \le a - r \le N$
slots ≥ 0
items ≥ 0 }

Recall:

Cons: process
⋮
P(items)
~~P(mutex)~~
val := bb[r mod N]
r := r+1
~~V(mutex)~~
V(slots)
end

items > 0 from P(items) implies $0 < a - r$ since items = a − r

{ $0 < a - r \wedge a - r \le N$ } implies

implies $a \bmod N \neq r \bmod N$! So delete P&V!