Een (OPERATING) SYSTEM in actie is een verzameling samenwerkende sequentiële processen, concurrent en asynchroon executerend, en gemeenschappelijke datastructuren manipulerend.

Asynchroon betekent dat de onderlinge verwerkingssnelheden non-deterministisch zijn.

Basisprobleem:

Wederzijdse uitsluiting

Voorbeeld:

```
program regelverwerker;
integer regelsin;

procedure proceseen;
begin
     herhaal eindeloos
          lees_een_regel;

          regelsin := regelsin + 1;

          verwerk_regel;
end;

procedure procestwee;
begin
     herhaal eindeloos
          lees_een_regel;

          regelsin := regelsin + 1;

          verwerk_regel;
end;

begin
     regelsin := 0;
     parbegin
          proceseen;
          procestwee;
     parend;
end.
```

Vraag: Welk probleem doet zich hier voor?

Een oplossing:

```
program regelverwerker;
integer regelsin;

procedure proceseen;
begin
    herhaal eindeloos
        lees_een_regel;

        StartWedUit;
        regelsin := regelsin + 1;

        StopWedUit;
        verwerk_regel;
end;

procedure procestwee;
begin
    herhaal eindeloos
        lees_een_regel;

        StartWedUit;
        regelsin := regelsin + 1;
        StopWedUit;

        verwerk_regel;
end;

begin
    regelsin := 0;
    parbegin
        proceseen;
        procestwee;
    parend;
end.
```

Dus: Processen moeten elkaar wederzijds uitsluiten als ze bijvoorbeeld gemeenschappelijke datastructuren manipuleren, in hetzelfde bestand schrijven, enz.

Noodzakelijk:

executie in Kritieke Sectie

Een Kritieke Sectie is
een verzameling instructies
vooraf gegaan door een StartWedUit
en afgesloten door een StopWedUit.

StartWedUit en StopWedUit moeten ondeelbare acties zijn !

De opdracht wordt dus:

Zoek oplossingen voor StartWedUit en StopWedUit die:

- **in software te realiseren zijn,**
- **onafhankelijk zijn van de onderlinge snelheden van processen,**

en daarbij:

- **wederzijdse uitsluiting garanderen,**
- **dodelijke omarming uitsluiten,**
- **uithongering uitsluiten,**
- **slechts zwakke samenhang toestaan.**

Opmerkingen

Nederlands	Engels
Wederzijdse uitsluiting:	Mutual exclusion
WedUit:	Mutex
Dodelijke omarming:	Deadlock
Uithongering:	Starvation Lockout
Zwakke samenhang:	Loose connection
Zwak samenhangend:	Loosely coupled

We beperken ons tot twee processen.
We formuleren de oplossingen mbv de concurrente statements.

Algemene structuur:

```
program probeer;

globale datastructuren;
declaratie proceseen;
declaratie procestwee;

begin
    parbegin
        proceseen;
        procestwee;
    parend;
end.
```

Algoritme 1:

Een beurt-indicator geeft aan welk proces de ks mag betreden.

```
-----------------------------------------------------
        Zolang Beurt != i Doe niets;
        kritieke sectie;
        Beurt := j;
-----------------------------------------------------
```

<u>Analyse</u>:

WedUit: ok, beurtelings

DodOma: processen beurtelings in ks

UitHon: processen beurtelings in ks

ZwaSam: niet ok, beurtelings

Algoritme 2:

Geef elk proces een eigen (beurt)indicator.

```
Zolang Vlag[j] = IN Doe niets;
Vlag[i] := IN;

kritieke sectie;

Vlag[i] := Niet_IN;
```

<u>Analyse:</u>

WedUit: niet gegarandeerd,
Voorbeeld scenario:
t0: P0 vindt Vlag[1] Niet_IN
t1: P1 vindt Vlag[0] Niet_IN
t2: P1 zet Vlag[1] op IN en betreedt ks
t3: P0 zet Vlag[0] op IN en betreedt ks

DodOma: ntz

UitHon: ntz

ZwaSam: ok

!!! Het probleem is blijkbaar:
de gescheiden test- en zet-acties.

Algoritme 3:

Probleem bij 2 was: eerst test, dan zet.
Probeer nu: eerst zet, dan test.

```
Vlag[i] := IN;
Zolang Vlag[j] = IN Doe niets;

kritieke sectie;

Vlag[i] := Niet_IN;
```

Analyse:

WedUit: ok

DodOma: mogelijk,
Voorbeeld scenario:
t0: P0 zet Vlag[0] op IN
t1: P1 zet Vlag[1] op IN

UitHon: ntz

ZwaSam: ok

Een eenvoudige oplossing lijkt moeilijk te vinden.
Oplossingen bestaan wel.
Eén van die oplossingen is een combinatie van algoritme 1 en algoritme 3, waarbij processen 'galant' zijn.

Algoritme 4:

Gebruik een indicator per proces en een gemeenschappelijke beurt-indicator.

```
-----------------------------------------------------
    Vlag[i] := IN;
    Beurt := j;(* wees galant*)
    Zolang (Vlag[j] = IN & Beurt = j) Doe niets;
    kritieke sectie;
    Vlag[i] := Niet_IN;
-----------------------------------------------------
```

<u>Analyse:</u>

WedUit: ok, want als Vlag[i]=Niet_IN, dan mag Pj in ks (en vv); als Vlag[i]=IN & Vlag[j]=IN, dan Beurt= i òf j (en niet beide); dus bv Pi passeert de Zolang en Pj lust, totdat Pi Vlag[i] op Niet_IN zet.

DodOma: ok, want als het ene proces hangt in de Zolang, dan kan het andere proces de Zolang passeren.

UitHon: ok, want als Pj "Zolang"t, dan Pi in ks. Als Pi uit ks, dan Vlag[i] := Niet_IN, dus Pj mag ks in; als Pj nu zodanig wordt opgehouden dat Pi weer gaat "Zolang"en, dan zijn beide Vlaggen IN en Beurt = j, dus blijft Pi "Zolang"en tot Pj geweest is.

ZwaSam: aanvaardbaar.

Terugkerend probleem:

Het niet in één (ondeelbare) actie kunnen testen èn zetten van een variabele.

Daarom: Hardware oplossing.

"test-en-zet"instructie:
Was = Test_en_Zet(&Variabele)

```
int Test_en_Zet(int *Bestemming)
{   int hulp;
    hulp = *Bestemming;
    *Bestemming = 1;
    return hulp;
}
```

"verwissel"instructie: Verwissel(&A, &B)

```
void Verwissel(int *A, int *B)
{   int hulp;
    hulp = *A;
    *A = *B;
    *B = hulp;
}
```

Gebruik bij kritieke sectie

Test_en_Zet:

```
Zolang Test_en_Zet(&Slot) != 0 Doe niets;

 kritieke sectie;

Slot := 0;
```

Verwissel:

```
Sleutel := 1;
Herhaal Verwissel(Slot, Sleutel) Tot Sleutel = 0;

 kritieke sectie;

Slot := 0;
```

Een oplossing voor N processen zonder uithongering

Gemeenschappelijke datastructuren:

```
    int Slot;
    int Wachtend[N];
```

Per proces Pi:

```
    int Sleutel;
    int j;
```

StartWedUit en StopWedUit:

```
Wachtend[i] := 1;
Sleutel := 1;
Zolang Wachtend[i] = 1 & Sleutel = 1
Doe Sleutel := Test_en_Zet(&Slot);
Wachtend[i] := 0;
```

kritieke sectie;

```
j := (i + 1) Mod N;
Zolang (j != i) & (Wachtend[j] = 0)
Doe j := (j + 1) Mod N;
Als j = i  Dan    Slot := 0;
           Anders Wachtend[j] := 0;
```

Analyse:

WedUit: ok

DodOma: ok

UitHon: ok, want
Als Pi StopWedUit bereikt, dan wachten één of meer processen(1) of er wacht geen proces(2).
(2): Slot := 0, dus geen uithongering;
(1): van het eerste proces in de 'nummer'volgorde [0..N-1] wordt Wachtend 0 terwijl Sleutel 1 blijft (en ook Slot), dit proces mag de Zolang verlaten.

Alle tot nu toe gepresenteerde oplossingen maken gebruik van

Actief wachten (Busy waiting)

Andere oplossingen zijn dus gewenst.

Vraag: Wanneer wordt een proces onderbroken?

Antwoord: Als een interrupt optreedt!

Mogelijke oplossing:
houd interrupts tegen gedurende de uitvoering van een kritieke sectie.

Nadelen:
- **interrupts kunnen verloren gaan**
- **verlaging graad van multiprogrammering**
- **responsetijd???**

enz.

Realistisch is wel het tegenhouden van interrupts gedurende zeer korte periodes.

Dus: **Creëer met behulp van het os (in system-mode) nieuwe ondeelbare operaties om StartMutex en StopMutex mogelijk te maken.**

Vermijd Actief-wachten door gebruik te maken van de procesmanipulatiemogelijkheden van het os.

Hiertoe is in 1965 door E W Dijkstra de Semafoor en het bijbehorend mechanisme ontwikkeld. Het woord Semafoor is een ander woord voor Seinpaal.

Beschrijving van de Semafoor

Een Semafoor is een object, dwz een datastructuur en enige operaties op die datastructuur.

Datastructuur
Struct sema { Int waarde; List of process L;}; Typedef struct sema Semafoor; Semafoor S;

Operaties	met hun functionaliteit
Wait(S)	S.waarde := S.waarde - 1; Als S.waarde < 0 Dan zet dit proces in S.L en blokkeer;
Signal(S)	S.waarde := S.waarde + 1; Als S.waarde <= 0 Dan haal proces uit S.L en maak ready;
sinit(S,i)	Als i < 0 Dan fout; S.waarde := i; initialiseer L;

NB: De organisatie van de lijst is niet vastgelegd.

NB: Aliassen voor Wait zijn Down (Tanenbaum) en de oorspronkelijke naam P ("passeren").
Aliassen voor Signal zijn Up (Tanenbaum) en de oorspronkelijke naam V ("vrijgeven v h traject").

Gebruiksmogelijkheden

- Mutex voor kritieke secties

- Synchronisatie van processen

Vb van een toepassing van een semafoor tbv mutex: Een concurrent programma waarin 5 processen 3 afdrukkers moeten delen. De uitvoer van de verschillende processen moet gescheiden blijven.

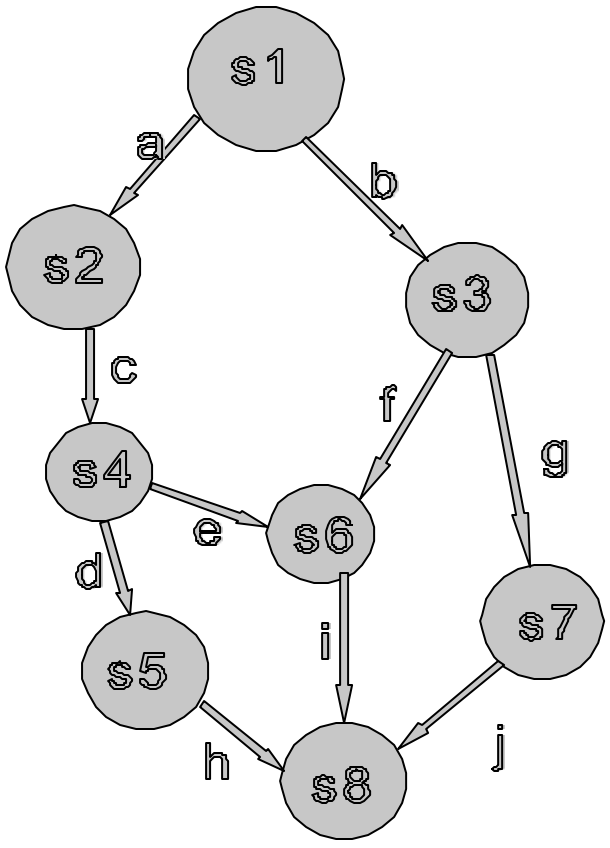
```
Program Afdruk_processen;
Semafoor Print;

Procedure P_1;
    .
    .
Procedure P_i;
Begin
  Herhaal
    doe_iets i;
    Wait(Print);
    bepaal welke afdrukker;
    doe het afdrukwerk;
    markeer afdrukker vrij;
    Signal(Print);
    rest i;
  Tot klaar;
End;
    .
    .
Procedure P_5;

Begin
  sinit(Print, 3);
  Parbegin
    P_1; P_2; P_3; P_4; P_5;
  Parend;
End.
```

Vb van het gebruik van een Semafoor ter synchronisatie van processen:
Proces_2 mag actie S2 pas uitvoeren als Proces_1 actie S1 voltooid heeft.
De Semafoor Sync wordt geïntialiseerd op 0.

```
Proces_1                Proces_2
.                           .
.                           .
S1                          P(Sync);
V(Sync);                S2
.                           .
.                           .
```

Nog een voorbeeld:

Een precedentiegraaf



Het concurrent programma

```
Semafoor a,b,c,d,e,f,g,h,i,j;
      alle geïnitialiseerd op 0

Parbegin
  Begin
      s1;V(a);V(b);
  end;
  Begin
      P(a);s2;V(c);
  end;
  Begin
      V(c);s4;V(d);V(e);
  end;
  Begin
      P(b);s3;V(f);V(g);
  end;
  Begin
      P(d);s5;V(h);
  end;
  Begin
      P(e);P(f);s6;V(i);
  end;
  Begin
      P(g);s7;V(j);
  end;
  Begin
      P(h);P(i);P(j);s8;
  end;
Parend;
```

Optimaal?:

```
Parbegin
      Begin s1;vb;s2;s4;ve;s5;pi;pj;s8 end;
      Begin pb;s3;vf;s7;vj end;
      Begin pe;pf;s6;vi end;
Parend;
```

Voorbeeld met Semaforen:
producent, consument (Xinu conventie).

```
#define PROCESS void
#define BUFLEN 100
#define MAXINT 32767
#define STK 512
#define SEMAPHORE int
PROCESS producer();
PROCESS consumer();

int buffer[BUFLEN], ppid, cpid;
SEMAPHORE produced, consumed;

PROCESS xmain()
{
    consumed=screate(BUFLEN);
    produced=screate(0);
    resume(ppid =create(producer, STK, 10, "p", 0));
    resume(cpid =create(consumer, STK, 10, "c", 0));
}
/*******************************************************
* functie producer: produceert integers
*******************************************************/
PROCESS producer()
{   int i;
    for ( i = 0; i <= MAXINT; i++) {
        wait(consumed);       /* P op 'consumed'  */
        buffer[i%BUFLEN] = i; /* plaats in buffer */
        signal(produced);     /* V op 'produced'  */
    }
}
/*******************************************************
* functie consumer: consumeert integers
*******************************************************/
PROCESS consumer()
{   int i, m;
    for ( i = 0; i <= MAXINT; i++) {
        wait(produced);        /* P op 'produced' */
        m = buffer[i%BUFLEN];  /* haal uit buffer */
        signal(consumed);      /* V op 'consumed' */
    }
}
```

Samenvatting Semafoor

Semafoor is een uitstekend mutex- en synchronisatie-middel op 'laag' niveau.

Semafoor is bij gebruik foutgevoelig:

- **Bij elke Wait hoort een Signal,**
  - **vergeten**
  - **verwisselen**

- **Geneste Semaforen gevaarlijk.**

Daarom verdienen 'hogere' constructies de voorkeur!

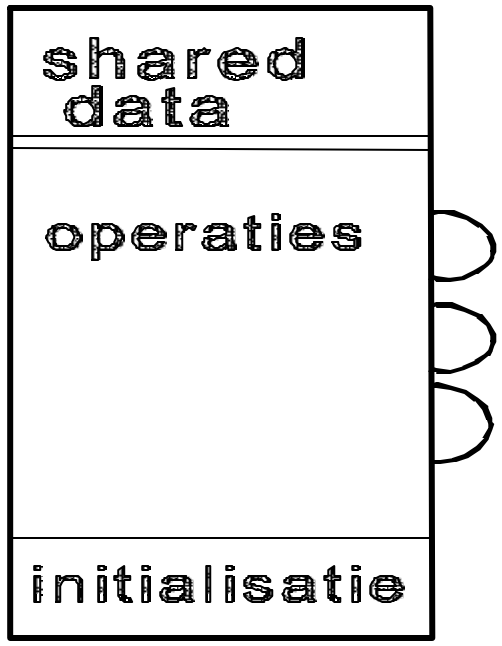
Klasse:
lokale data +
operaties +
initialisatie

De gebruiker kent slechts het bestaan van dit object en kan het gebruiken door de betrokken operaties uit te voeren. (vergelijk Definitie Module van Modula.)

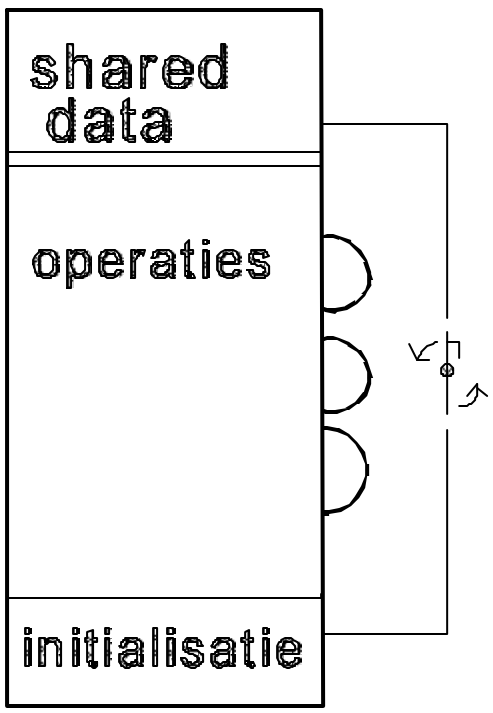
De Klasse staat geen concurrentie toe.

Oorsprong van de Klasse: Simula 67

Klasse



Klasse + Mutex



Hoe kan de Klasse omgevormd worden zodat hij geschikt wordt voor concurrente omgevingen?

Door de operaties onder mutex te laten uitvoeren?

Wat doen we dan met een proces dat moet wachten in de Klasse?

Vb: een Klasse die een buffer implementeert: een producent moet wachten omdat de buffer vol is, totdat een consument ruimte heeft gemaakt.

Voorbeeld: Eindige buffer

Naam van de monitor	monitor eindige_buffer;
Shared data	int buffer[lengte]; int in, uit; int n;

Operatie: plaats in buffer
 	
Operatie: haal uit buffer
	void plaats(int v);
	{ if (n == lengte) "WACHT OP NIET VOL";
	buffer[in] = v; in++; n++;
	if (in == lengte) in = 0;
	"SIGNALEER BUFFER NIET LEEG";
	}
	
	void haal(int *pv);
	{ if (n == 0) "WACHT OP NIET LEEG";
	*pv = buffer[uit]; uit++; n--;
	if (uit == lengte) uit = 0;
	"SIGNALEER BUFFER NIET VOL";
	}

Initiali-satie	in =0; uit = 0; n = 0;

Deze aangepaste Klasse wordt genoemd:

Monitor

Voordelen van de Monitor:

- Gemeenschappelijke datastructuren lokaal
- Variabelen lokaal
- Operaties op datastructuren slechts mogelijk mbv de monitor-routines, mutex gegarandeerd
- Bij inbouw in taal: compilatie-bescherming
- Overzichtelijke constructie
- Eenvoudiger Wait en Signal dan bij Semaforen omdat de tel- en blokkeer-functies gescheiden kunnen worden

Vraag: Wat te doen met de Monitor-Wait en -Signal?

We noemen deze resp. Delay en Continue.

Antwoord: Creëer een nieuw type: Conditie

De enig mogelijke operaties op een variabele van het type Conditie zijn:

- **Delay(c): Het aanroepend proces wordt in de wachtrij gezet die behoort bij deze conditie c. De organisatie van de wachtrij is FIFO.**

- **Continue(c): Als wachtrij van conditie c leeg,**
  Dan doe niets,
  Anders wek het 1e proces in de rij op.

Consequenties:

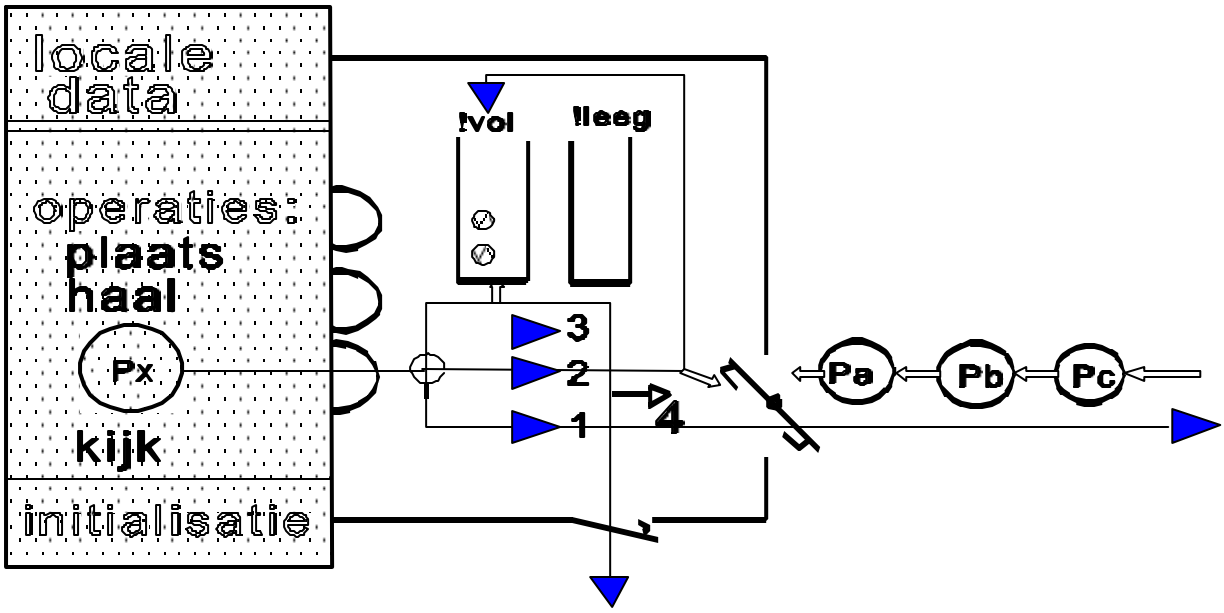
- **Delay moet de mutex op de Monitor opheffen anders is dodelijke omarming mogelijk.**
- **Continue mag slechts als laatste opdracht van een monitor-routine uitgevoerd worden, anders kunnen mutex-problemen ontstaan.**

Voorrang:

- **Processen wachtend in Delay hebben bij herstart (onder mutex) voorrang op processen die buiten de Monitor op toegang wachten.**

Een Monitor-type volgens een wat eenvoudiger concept als boven, is door Brinch-Hansen geïmplementeerd in zijn taal 'Concurrent Pascal'. Deze taal en drie operating systems worden beschreven in zijn boek 'The Architecture of Concurrent Programs' (P-H 1977).

MONITOR met 2 condities



1 Px heeft een Kijk uitgevoerd en vertrekt door de voordeur;

2 Px probeert een Plaats uit te voeren maar de buffer is vol, dus Px gaat in de wachtrij voor conditie !vol na de voordeur geopend te hebben;

3 Px heeft een Haal uitgevoerd, laat een proces toe dat wacht op !vol en vertrekt door de zijdeur;

(4 Als bij 3 de wachtrij leeg blijkt te zijn, dan vertrekt Px door de voordeur.)

De gereedschappen Semafoor en Monitor zijn uitwisselbaar.

Een voorbeeld:

Simulatie van een binaire semafoor mbv een monitor

```
Monitor semsim;
var bezig : boolean;
    niet_bezig : condition;

procedure entry Down;
begin
    if bezig then
    Delay(niet_bezig);
    bezig := true;
end;

procedure entry Up;
begin
    bezig := false;
    Continue(niet_bezig);
end;

begin (* initialisatie *)
    bezig := false;
end;
```

Vraag: Tellende semafoor?

Simulatie van een Monitor mbv Semaforen.

Monitor-mutex: (binaire) semafoor (init 1)

Conditie: (binaire) semafoor (init 0)
+ teller wachtenden (init 0)

Stel de mutex-semafoor heet Mutex,
en de conditie-semafoor heet Cond met bijbehorende aparte teller CondTeller

dan:

elke Monitor-procedure begint met:

```
wait(Mutex)
```

Delay(conditie) wordt:

```
CondTeller := CondTeller + 1;
signal(Mutex);
wait(Cond);
CondTeller := CondTeller - 1;
```

als de procedure eindigt met Continue(conditie)

dan

```
if CondTeller > 0
    then  signal(Cond)
    else  signal(Mutex);
```

anders

```
signal(Mutex)
```

Hierna volgt een voorbeeld van een gesimuleerde Monitor die een eindige buffer implementeert tbv producenten en consumenten (Xinu conventie).

```
#include <alias.h>
#define BUFLEN 10
local char buffer[BUFLEN][80];
local SEMAPHORE
    mutex, wacht_op_geleegd, wacht_op_gevuld;
local int lege_plaatsen, volle_plaatsen,
                 inwijzer, uitwijzer;
local int wachtenden_op_leeg, wachtenden_op_vol;

void init()
{ mutex = screate(1);
  wacht_op_geleegd = screate(0);
  wacht_op_gevuld = screate(0);
  lege_plaatsen = BUFLEN;
  volle_plaatsen = inwijzer = uitwijzer = 0;
  wachtenden_op_leeg = wachtenden_op_vol = 0;
}
void plaats(char *in)
{ down(mutex);
  if (volle_plaatsen == BUFLEN){
    wachtenden_op_leeg++;
    up(mutex);
    down(wacht_op_geleegd);
  }
  lege_plaatsen--; volle_plaatsen++;
  strcpy(buffer[inwijzer++], in);
  if (inwijzer >= BUFLEN) inwijzer = 0;
  if (wachtenden_op_vol){
    wachtenden_op_vol--;
    up(wacht_op_gevuld);
  }
  else up(mutex);
}
void haal(char *uit)
{ down(mutex);
  if (lege_plaatsen == BUFLEN){
    wachtenden_op_vol++;
    up(mutex);
    down(wacht_op_gevuld);
  }
  lege_plaatsen++; volle_plaatsen--;
  strcpy(uit, buffer[uitwijzer++]);
  if (uitwijzer >= BUFLEN) uitwijzer = 0;
  if (wachtenden_op_leeg){
    wachtenden_op_leeg--;
    up(wacht_op_geleegd);
  }
  else up(mutex);
}
```

- Mutex
- Synchronisatie

- Communicatie

via:

- gemeenschappelijk geheugen
- boodschappen

system calls:

- Send
- Receive