Betriebssysteme

Kap. 3: Interprozesskommunikation (IPC)

3.2: Höhere IPC Konzepte

Stand: WS 12/13 (22.01.13)

Prof. Dr. Wolfgang Küchlin

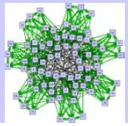
Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

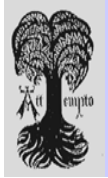
Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

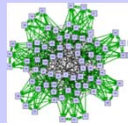
Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

SR 

Teil II: Höhere IPC Konzepte

- Kooperation: das Producer-Consumer Problem
- Conditions
- Semaphore
- Readers/Writer Locks
- Message Passing
- Monitore

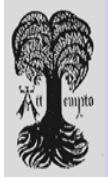


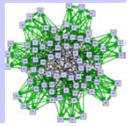


Höhere Konzepte zur Synchronisation

- Von BS zur Verfügung gestellte (höhere) Konzepte
  - Mutex/Conditions (POSIX Threads)
  - Semaphore (UNIX-System V und Nachfolger, FreeBSD 5.2)
  - Monitore (Concurrent Pascal, Modula-2, Java)

- Einzig voll portable Lösung (ursprüngliches UNIX)
  - Verwendung eines Lock-Files
  - Üblicherweise garantiert das BS, dass ein Fehler auftritt, falls das File bereits existiert.
  - Lock-Files werden bei Programmende/Absturz nicht automatisch gelöscht
  - Langsam, verursacht großen Overhead.





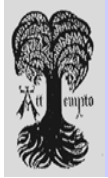
Mutex für POSIX-Threads

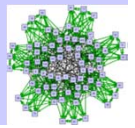
P1:

```
while ( ... ) {
   :
   pthread_mutex_lock(&mx);
   :   (krit. Bereich)
   pthread_mutex_unlock(&mx);
   :
}
```

P2:

```
while ( ...) {
   :
   :
   p...lock(&mx);
   :  (krit. Bereich)
   p...unlock(&mx);
   :
}
```

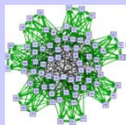




Mutex für POSIX-Threads

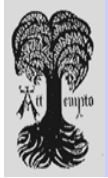
```
pthread_mutex_t mutex;
int pthread_mutex_init (mutex,
                        pthread_mutex_attr_t attr);
int pthread_mutex_destroy (mutex);
int pthread_mutex_lock (mutex);
int pthread_mutex_unlock (mutex);
int pthread_mutex_trylock (mutex);
```

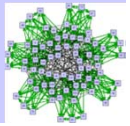
- Unterschiede zu Software-Lösung mit TSL:
  - BS blockiert anfragenden Thread falls Lock belegt
  - BS stellt Fairness mit Queue sicher
  - Involviert System Call
- Solaris
  - Anfragender Thread wird nur blockiert, falls Thread mit Lock nicht gerade läuft (d.h. auf 1-Proz. immer)



Inaktives Warten

- Beispiel für Kooperation: Producer/Consumer
  - Warten bei leerem/vollem Puffer dauert i.A. zu lange für busy waiting.
  - zwei neue Systemaufrufe:
    - SLEEP() :blockiert den Aufrufer, so dass er von Arbeit suspendiert wird, ohne einen Prozessor zu belasten.
    - WAKEUP(pid) : weckt den Prozess mit der ID pid auf.
  - alternativ: Einführung einer Ereignisvariable event
    - WAIT(event)
    - SIGNAL(event)

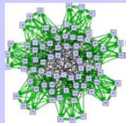




Producer/Consumer – 1.Näherung

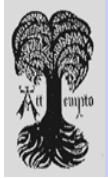
```
void producer()
{  int item;
   while (1)
   {  produce_item (& item);
      if (count == N)
         sleep(); /* if buffer full */
      enter(item);
      count = count + 1;
      if (count == 1)
         wakeup(consumer);
         /* if buffer was empty,
            hence consumer slept */
   }
}
```

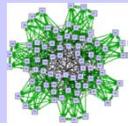
```
void consumer()
{  int item;
   while (1)
   {  if (count == 0)
         sleep();
         /* if buffer empty */
      remove (& item);
      count = count - 1;
      if (count == N-1)
         wakeup(producer);
         /* if buffer was full,
            hence producer slept */
      consume (item);
   }
}
```



Producer/Consumer – 1.Näherung

- Näherung inkorrekt
- Probleme:
  - Ausfall-Problem bei der Variable count
  - Konsistenz-Problem zwischen dem tatsächlichen Pufferzustand und dem Wert der Variablen count

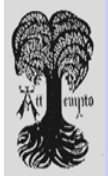


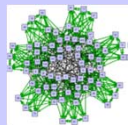


Producer/Consumer – 2.Näherung

```
mutex_t lock;
void producer()
{  int item;
   while (1)
   {  produce(& item);
      mutex_lock(lock);
      if (count == N) {
         mutex_unlock(lock);
         sleep();
         mutex_lock(lock);
      }
      enter(item);
      count = count + 1;
      if (count == 1)
         wakeup(consumer);
      mutex_unlock(lock);
   }
}
```

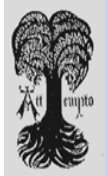
```
void consumer()
int item;
{  while (1)
   {  mutex_lock(lock);
      if (count == 0) {
         mutex_unlock(lock);
         sleep();
         mutex_lock(lock);
      }
      remove(& item);
      count = count - 1;
      if (count == N-1)
         wakeup(producer);
      mutex_unlock(lock);
      consume(item);
   }
}
```

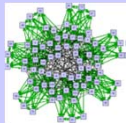




Producer/Consumer – 2.Näherung

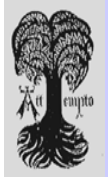
- 2. Näherung auch inkorrekt
- Bei Unterbrechung zwischen mutex_unlock() und sleep() schlafen bald beide Prozesse:
  - wakeup()-Aufruf des Konsumenten verpufft
- Mögliche Verbesserung:
  - if (count > 0) wakeup(consumer)
  - if (count < N) wakeup(producer)
  - geringere Wahrscheinlichkeit für Fehler
  - trotzdem Verletzung von Bedingung 2
- Lösungen
  - Man kann das Aufschließen und Einschlafen in einem Befehl kombinieren, wogegen
  - Dijkstras *Semaphore* die gesendeten Signale zählen, damit keines verloren geht.

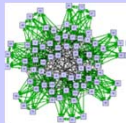




Conditions API

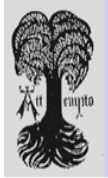
- *Conditions* sind die Realisierung der Ereignisvariablen in C Threads und Pthreads.
- Jedes Speicherobjekt ist typischerweise durch ein `mutex` geschützt (Synchronisation).
- Ein Thread erwirbt das Zugriffsrecht, untersucht das Objekt und findet, dass er auf eine Bedingung warten muss.
  - Das Zugriffsrecht muss abgegeben werden, damit ein anderer Thread die Daten verändern kann, bis die Bedingung erfüllt ist. (Kooperation)
- Das Abgeben des Zugriffsrechts geschieht atomar
  - `mutex` wird erst (vom System) freigegeben, wenn der Thread schläft und das Aufwach-Signal empfangen kann.

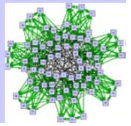




Conditions API

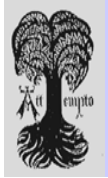
- Nachdem der blockierte Thread das Signal empfangen hat, wacht er mit dem Zugriffsrecht in der Hand auf.
  - Der Thread ist in dem Zustand, in dem er zuletzt war.
  - Er weiß jetzt, dass ein Anderer signalisiert hat, dass die Bedingung jetzt erfüllt ist.
  - Er muss trotzdem die Bedingung erneut prüfen, denn ein Dritter, der zusammen mit ihm auf die Erfüllung der Bedingung gewartet hat, könnte ihm zuvorgekommen sein und die Daten schon verändert haben, sodass die Bedingung schon wieder nicht mehr gilt und er erneut warten muss.
  - Bei nur 2 beteiligten Threads gibt es keinen solchen Dritten
  - Wird die Bedingung nicht erneut geprüft, ist Anforderung 2 verletzt (sobald ein dritter Thread hinzukommt, entsteht eine Fehlerquelle)

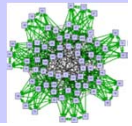




Diskussion zum Einsatz von Conditions

- Synchronisationsfehler treten nur sporadisch und nicht reproduzierbar auf
  - Kunde kauft mehr Prozessoren
  - Kunde konfiguriert mehr Threads
  - Anwendung produziert Crash im Weihnachtsgeschäft

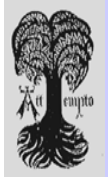


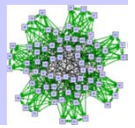


Pthreads / C Threads Conditions API

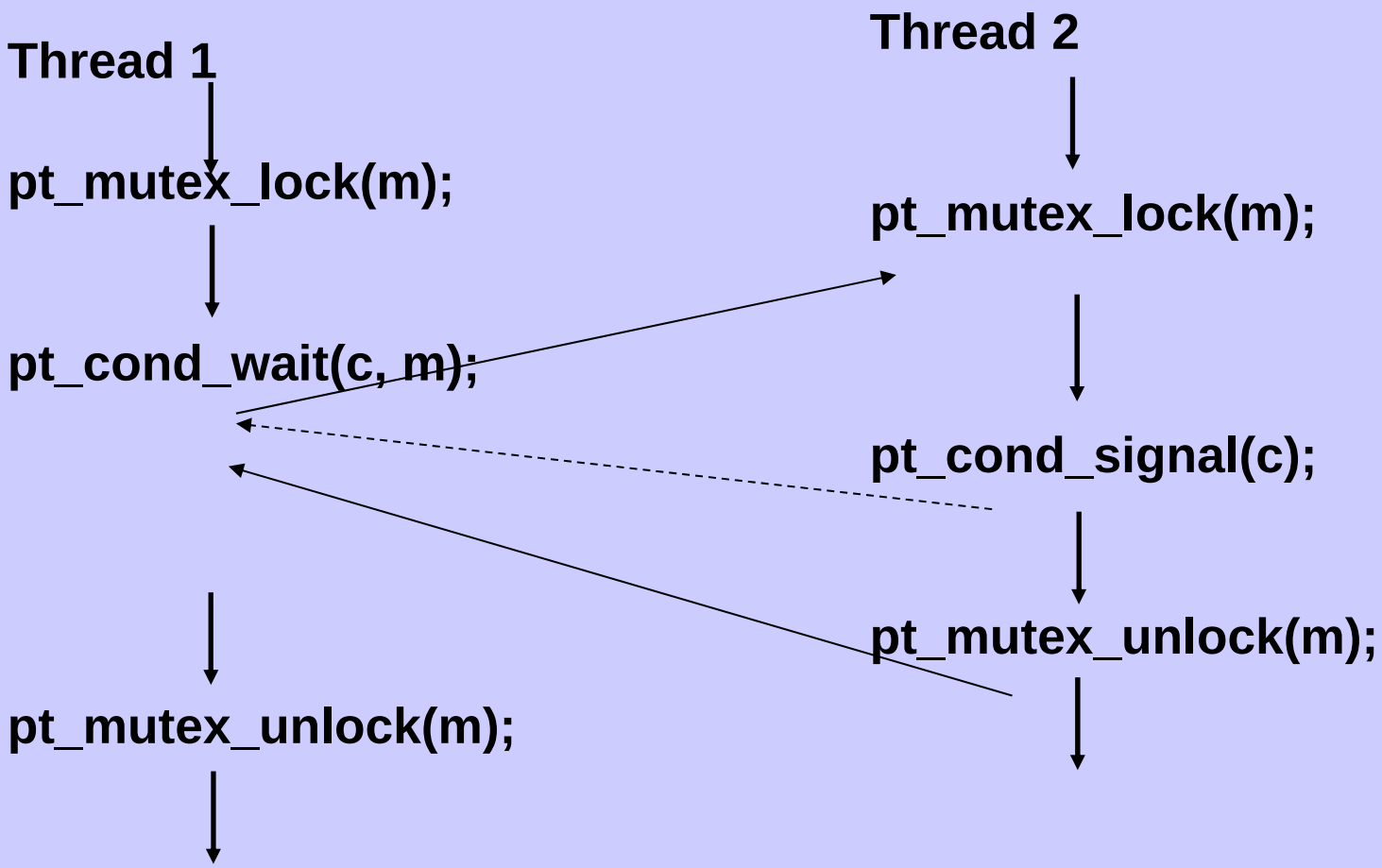
- pthread_cond_wait(cond, m) gibt Mutex m implizit frei und blockiert, bis cond signalisiert wurde.
- pthread_cond_signal(cond) deblockiert einen Thread, der auf cond wartet.
- pthread_cond_broadcast(cond) deblockiert alle Threads (aber nur einer bekommt das Mutex!)
- C Threads API:

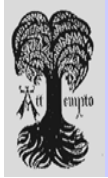
```
void condition_signal(c)
        condition_t c;

void condition_wait(c,m)
        condition_t c;
        mutex_t m;
```

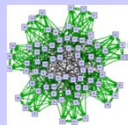




Mutex und Conditions: Typischer Ablauf







1 Producer / 1 Consumer

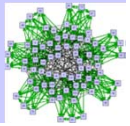
Initialisierung: cond_init(cond), mutex_init(m), n=0

Producer:

```
produce(&item);
mutex_lock(m);
if (n == bufsize)
    cond_wait(cond, m);
put(item);
n++;
cond_signal(cond);
mutex_unlock(m);
```

Consumer:

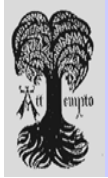
```
mutex_lock(m);
if (n == 0)
    cond_wait(cond, m);
get(&item);
n--;
cond_signal(cond);
mutex_unlock(m):
consume(item);
```

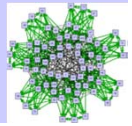


Spezialfall: 1 Producer / 1 Consumer

- Hier bedeutet die Condition c:

  „Ein Modifikationsereignis ist eingetreten“
- Bei 1 Producer + 1 Consumer alternieren die Ereignisse
  - Falls der Producer gewartet hatte, gilt bei 1 Producer Modifikation = Konsum
  - Falls der Consumer gewartet hatte, gilt bei 1 Consumer Modifikation = Produktion
- Erneute Prüfungen können theoretisch wegfallen (deshalb hier if statt while)





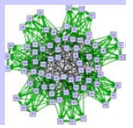
Allgemeinfall: n Producer / n Consumer

- Auch hier bedeutet eine einzige Condition c:

  „Ein Modifikationsereignis ist eingetreten“
- Bei n Producer + n Consumer alternieren die Ereignisse nicht mehr unbedingt
  - Konsum-Ereignisse und Produktions-Ereignisse nicht unterscheidbar
  - Ein wartender Producer kann durch nochmalige Produktion geweckt werden, ein wartender Consumer durch nochmaligen Konsum
- Erneute Prüfung der Bedingung nötig
- Unterscheidung der Ereignisse „Produktion“ und „Konsum“

  (aus Effizienzgründen)
  - Producer wartet nur auf Konsum-Ereignis, Consumer nur auf Produktions-Ereignis
  - Erneute Prüfung der Bedingung immer noch nötig

    (Konkurrenz in den Warteschlangen)



n Producer / n Consumer

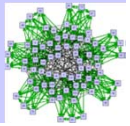
Initialisierung: mutex_init(m), n=0,
cond_init(nonfull), cond_init(nonempty)

Producer:

```
produce(&item);
mutex_lock(m);
while (n == bufsize)
    cond_wait(nonfull, m);
put(item);
n++;
cond_signal(nonempty);
mutex_unlock(m);
```

Consumer:

```
mutex_lock(m);
while (n == 0)
    cond_wait(nonempty, m);
get(&item);
n--;
cond_signal(nonfull);
mutex_unlock(m):
consume(item);
```



POSIX API für Conditions

pthread_cond_init()

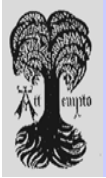
pthread_cond_timedwait()

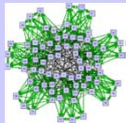
pthread_cond_broadcast()

pthread_cond_wait()

pthread_cond_signal()

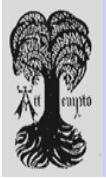
pthread_cond_destroy()

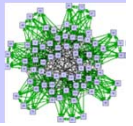




Semaphore

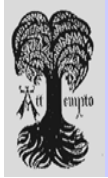
- Semaphor-Konzept wurde von E. W. Dijkstra erfunden.
- ursprünglicher Sinn:
  - Semaphor reguliert die Anzahl der Prozesse, die sich in einer KR befinden können bzw. die Anzahl der Exemplare einer KR, die verfügbar sind.
  - Das Semaphor wird mit dieser Zahl initialisiert.
- Es gibt 2 Operationen
  - **P(s):** „Passiere“ den Eingang zur KR (**p**ass).
  - **V(s):** „Verlasse“ die KR (lea**v**e).

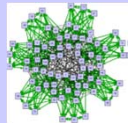




Semaphore

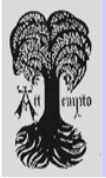
```
P(s):
  if (s > 0)
      s = s - 1;
  else
      blockiere den Prozess und
      trage ihn in die Queue von s ein

V(s):
   if (Queue von s leer)
      s = s+1;
  else
      entferne einen Prozess aus der Queue
      von s und wecke ihn dadurch
```

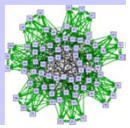




Semaphore

- Ein Semaphor kann als Kombination von Zähler und mutex betrachtet werden.
- Spezialfall: *binäres* Semaphor
  - Semaphor initialisiert zu 1
  - P(s) und V(s) sind äquivalent zu mutex_lock und mutex_unlock.
- Üblicherweise assoziiert man mit Semaphoren inaktives Warten.
  - Man benötigt aus Sicht von C Threads zusätzlich eine Bedingungsvariable, etwa non_empty
- Verteilung beschränkter Betriebsmittel:
  - P(s)=Down(s) vergibt ein Exemplar,
  - V(s)=Up(s) gibt ein Exemplar zurück.
- Ein Semaphor mit inaktivem Warten kann zur Realisierung von *Sleep, Wakeup* benutzt werden:
  - Down(e) wartet auf das Ereignis *e*,
  - Up(e) signalisiert *e*.
- Es gehen keine Weckrufe verloren, da sie von der Semaphor-Variable gezählt werden.





Producer/Consumer mit Semaphor

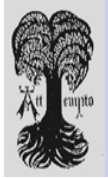
- Initialisierung
  - Semaphore full = 0 und empty = n (Größe des Buffers)
  - binäres Semaphor mutex = 1

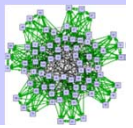
Producer:

```
produce(item);
    P(empty);
    P(mutex);
put(item);
    V(mutex);
    V(full);
```

Consumer:

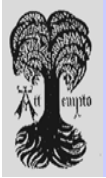
```
    P(full);
    P(mutex);
get(item);
    V(mutex);
    V(empty);
consume(item);
```

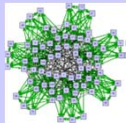




Semaphore – Producer/Consumer

- Initialisierung
  - empty = N
  - full = 0
- Semaphore empty und full werden als Bedingungsvariablen benutzt.
- Semaphor mutex wird als Schlossvariable benutzt.



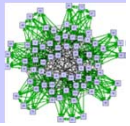


Semaphore – Producer/Consumer

synchron. Eintragen von item

synchron. Austragen von item

```
typedef int * semaphore; semaphore mutex, empty, full;
producer
{   produce(& item); /* Außerhalb des KA nebenläufig */
    down(empty);     /* Warte auf ein leeres Fach */
    down(mutex);     /* Zugangsrecht vergeben */
    enter(item);
    up(mutex);       /* Zugangsrecht zurückgegeben */
    up(full); /* Signalisiere Existenz eines vollen Fachs */
}

consumer
:
{   down(full);      /* Warte auf ein volles Fach */
    down(mutex);     /* Zugangsrecht */
    remove(& item);
    up(mutex);       /* Zugangsrecht zurück */
    up(empty); /* Signalisiere Existenz eines leeren Fachs */
    consume(item);   /* außerhalb des KA nebenläufig */
}
```



Semaphore

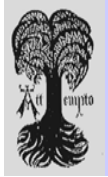
- Implementation eines allg. Semaphors S mit binären Semaphoren S_1 und S_2

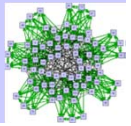
Initialisierung:
```
S1 = 1;
S2 = 0;
```

V(S):
```
P(S1);
S = S+1;
if ( S <= 0 ) {
   V(S2);
   V(S1);
} else
   V(S1);
```

P(S):
```
P(S1);
S = S-1;
if ( S<0 ) {
   V(S1);
   P(S2);
} else
V(S1);
```

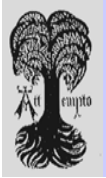
- Mit mutex_lock(L_i) für P(S_i) und mutex_unlock(L_i) für V(S_i) erhalten wir Implementation mit Mutex, Initialisierung mutex_lock(L_2)

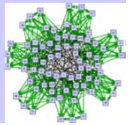




Semaphore & Conditions

- Unterschiede zwischen Semaphoren und Conditions
  - Condition Variable hat keinen Wert
  - cond_signal verpufft wirkungslos, falls kein Thread mit cond_wait wartet
    - V(s) speichert das Signal, bis es abgerufen wird
  - cond_wait blockiert immer
    - P(s) blockiert nur, falls Semaphor S=0

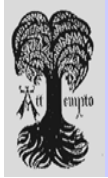


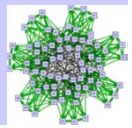


Readers-Writers Problem

- Ressource darf von beliebig vielen Prozessen gelesen, aber nur von einem (und dann ohne gleichzeitiges Lesen) geschrieben werden.
- Problem:
  - Falls Reader Priorität haben kann der Writer verhungern
- Modifikation:
  - Zusätzliche Schranke für neue Reader aktivieren, sobald ein Write-Zugriff gefordert wird. Reader sterben danach langsam aus.

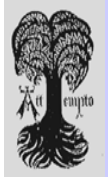
```
Reader:  lock(x);                Writer:  P(W);
         r=r+1;                           WRITE;
         if (r==1) P(W);                  V(W);
         unlock(x);
         READ;
         lock(x);
         r=r-1;
         if (r==0) V(W);
         unlock(x)
```

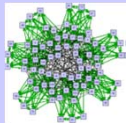




Message Passing

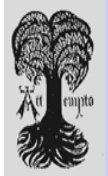
- Applikation besteht aus mehreren, getrennten Adressräumen (üblicherweise als UX-Prozesse realisiert).
- Kommunikation zwischen Adressräumen erfolgt mittels Nachrichtenaustausch
  - send / receive Primitive müssen vorhanden sein (s. u.).
- Konsequenzen:
  - Daten explizit den einzelnen Adressräumen zuordnen.
  - Jede Art von Interaktion erfordert explizite Kooperation der beteiligten Prozesse (Besitzer und Verwender der Daten).

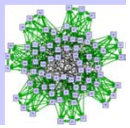




Message Passing

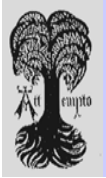
- Voraussetzung für bisherige Konzepte: gemeinsamer Speicher
  - Bei verteilten Systemen nicht unbedingt gegeben
- Jetzt: Prozesse schicken sich Nachrichten zu.
- Auch sinnvoll für Systeme mit gemeinsamem Speicher
  - besserer Speicherschutz gewährleistet
  - Philosophie von MACH mit *message ports* (Nachrichtenpforten) in jedem Prozess.
- Nachrichten können auf zwei Arten übermittelt werden:
  - synchron im *Rendezvous* synchron oder
  - asynchron, wobei sie in einem *Briefkasten* (mailbox) zwischengespeichert werden.
    - MACH *ports* sind vom BS verwaltete und geschützte Briefkästen.

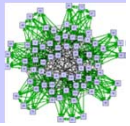




Message Passing

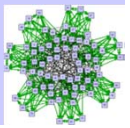
- Grundlegende Operationen
  - send (destination, message)
  - receive (source, message)
- Fragestellungen
  - Synchronisation
    - send entweder blocking oder nonblocking
    - receive entweder blocking oder nonblocking
- Fragestellungen
  - Adressierung
    - Direkt (explizit, implizit)
    - Indirekt (statisch, dynamisch, ...)
  - Format: Länge
  - Buffer-Verhalten
    - FIFO
    - Prioritäten





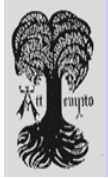
Client/Server-Modell – (un)gepufferte Aufrufe

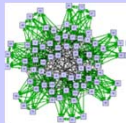
- *synchron* und *ungepuffert* → Rendezvous zwischen Sender und Empfänger
- *asynchron* und *gepuffert* → Kommunikation zeitlich entkoppelt (mailbox)
- asynchroner Aufruf
  - nur je ein Teil-Thread der Partner, der asynchron vom (Haupt-)Thread läuft, treffen sich genau
- gepufferter Aufruf
  - eingetroffene Nachricht wird vom BS zwischengespeichert, bis sie vom Empfänger abgerufen werden kann
- ungepuffertes `receive(addr,&buf)`
  - Empfänger wartet auf Nachricht von Adresse addr
  - Kommt Nachricht an, so wird receive() deblockiert und übernimmt sie in `buf`
  - Kein receive()-Aufruf → Nachricht geht verloren
- asynchrones `receive()`
  - receive()-Thread verwaltet Nachricht, bis Haupt-Thread sie braucht
  - Empfänger richtet zu Programmbeginn Briefkasten (*mailbox*) ein
  - BS speichert eingehende Nachrichten (bis zu max. Anzahl)



Send und Receive

- **`send(void* sendbuf, int nelems, int dest)`**
  - **`sendbuf:`** Zeiger auf Puffer mit den zu sendenden Daten
  - **`nelem:`** Anzahl der zu sendenden Datenworte
  - **`dest:`** ID des Empfänger-Prozesses
- **`receive(void* recvbuf, int nelems, int source)`**
  - **`recvbuf:`** Zeiger auf Puffer für die zu empfangenden Daten
  - **`nelem:`** Anzahl der zu empfangenden Datenworte
  - **`source:`** ID des Sender-Prozess





Blockierende vs. nicht-blockierende Message Passing Operationen

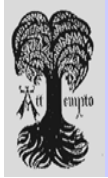
Prozess P0

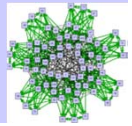
```
a = 100;
send(&a,1,1);
a = 0;
```

Prozess P1

```
receive(&a,1,0);
printf("%d\n",a);
```

- **Blockierende Message Passing Operationen**
  - Send Aufruf kehrt erst zurück, wenn der Sendepuffer modifiziert werden kann, ohne die Semantik zu verändern.
- **Nicht-blockierende Message Passing Operationen**
  - Send Aufruf kehrt sofort zurück; Korrektheit muss vom Programmierer sichergestellt werden.

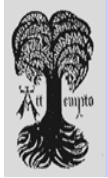


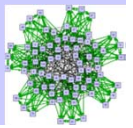


Blockierende nicht-gepufferte Send/Recv Ops

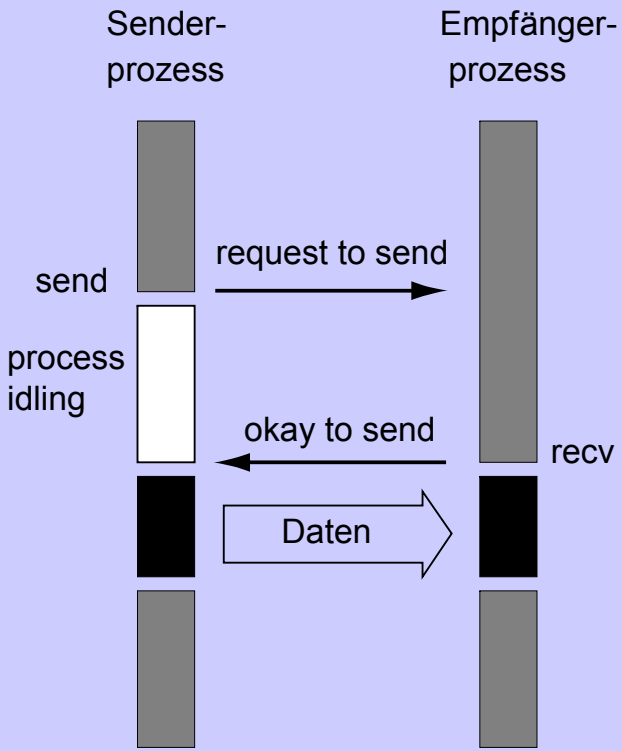
- Send-Aufruf kehrt erst zurück, wenn
  - der entsprechende Receive-Aufruf stattgefunden hat
  - und die Nachricht vollständig übertragen wurde.
- Vorteil: Effiziente Implementierung durch (R)DMA
- Nachteile:
  - Falls keine enge Synchronisation möglich, wird der Sender- oder Empfängerprozess blockiert (→ **Process Idling**).
  - Deadlocks möglich:

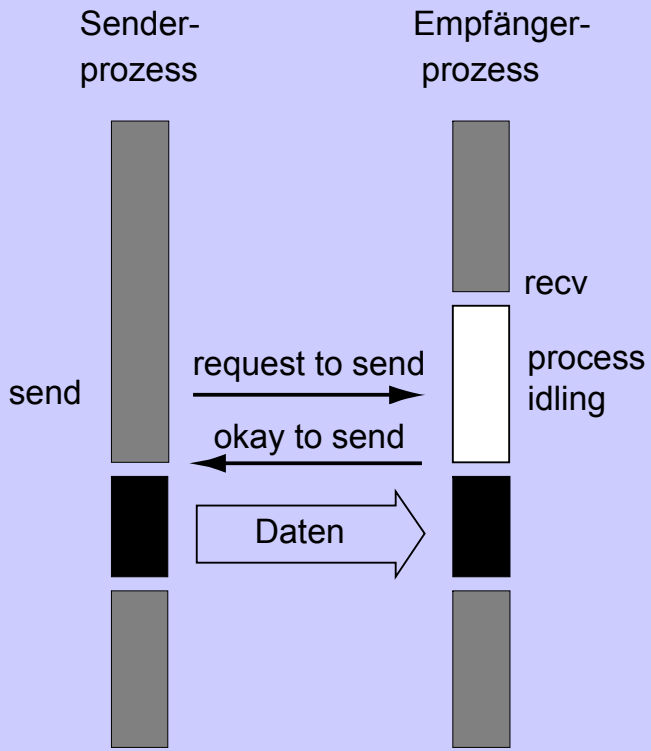
Prozess P0	**Prozess P1**
`send(&a,1,1);`	`send(&b,1,0);`
`receive(&b,1,1);`	`receive(&a,1,0);`

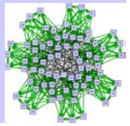




Blockierende nicht-gepufferte Send/Recv Ops.







Blockierende gepufferte Send/Recv Operationen

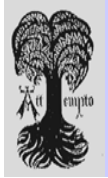
- Sender und/oder Empfängerprozess verwenden interne Pufferspeicher für die Kommunikation.
- Vorteil: Entkopplung von Sender und Empfänger.
- Nachteile:
  - Overhead für Puffermanagement (Kopieren der Daten, ...).
  - Deadlocks möglich (receive Operation kehrt erst zurück, wenn Daten im lokalen Puffer verfügbar sind):

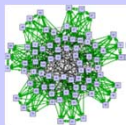
Prozess P0

```
receive(&a,1,1);
send(&b,1,1);
```

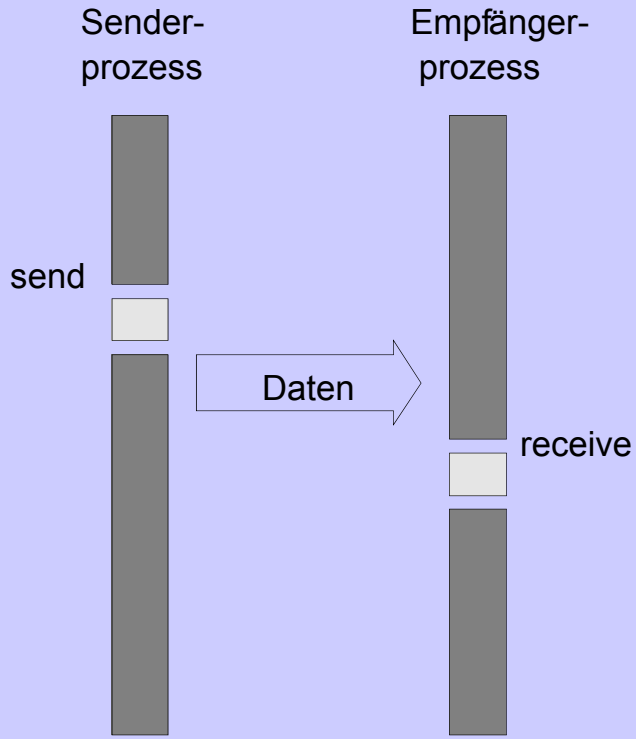
Prozess P1

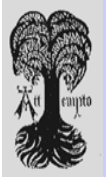
```
receive(&b,1,0);
send(&a,1,0);
```

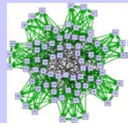




Blockierende gepufferte Send/Recv Operationen

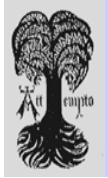


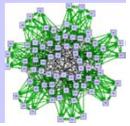




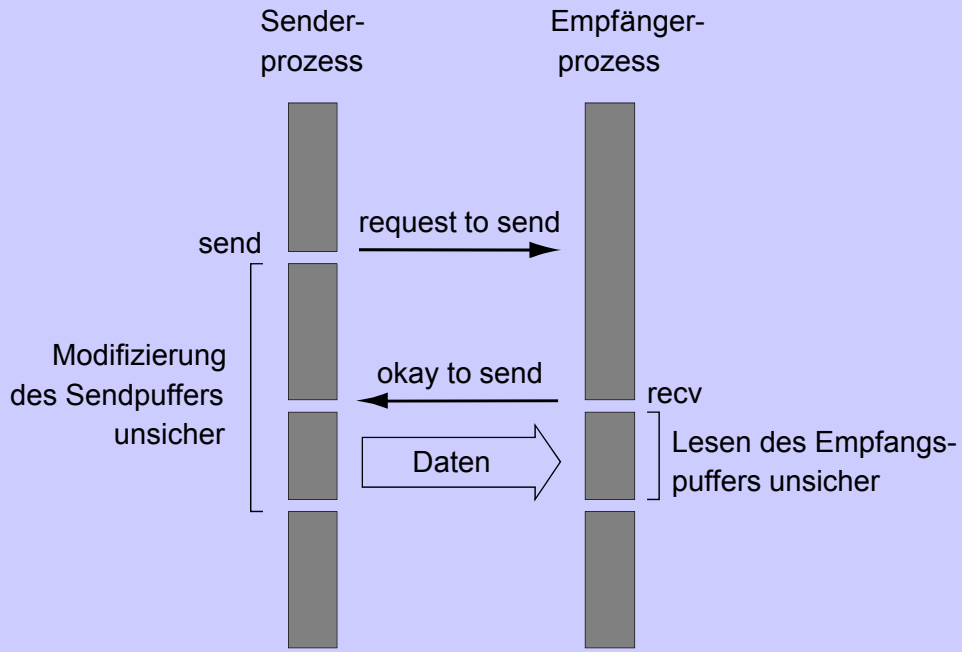
Nicht-blockierende Send/Receive Operationen

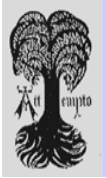
- Nicht-blockierende Send/Receive Aufrufe kehren zurück, bevor Puffervariablen sicher geändert werden können.
- Vorteil: Kein Overhead in Form von Process Idling oder Puffermanagement wie bei den blockierenden Operationen.
- Nachteil: Programmierer muss sicherstellen, dass Puffervariablen nicht vor Beendigung der Kommunikationsoperation verändert werden.
  - **Check-Status** Primitiv gibt Auskunft, ob Puffervariablen sicher überschrieben werden können.

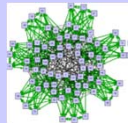




Nicht-blockierende Send/Receive Operationen

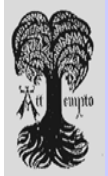


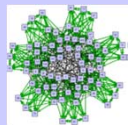




Synchrone und asynchrone Kommunikation

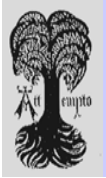
- Verhalten beim Senden
  - Blockieren bis zum Empfang
    - Speichern der Mitteilung im Buffer
- Verhalten beim Empfangen
  - Blockieren bis Mitteilung vorliegt
    - Return mit Fehlermeldung
- send und receive blockierend
  - Synchrone Kommunikation, Rendezvous Buffer unnötig
  - Asynchron
    - Entweder send oder receive (oder beide) nicht blockierend

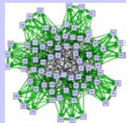




Synchrone und asynchrone Kommunikation

- send blockierend, receive nicht blockierend.
  Buffer unnötig.
- send nicht blockierend, receive blockierend.
  Häufigste Variante
- send und receive nicht blockierend





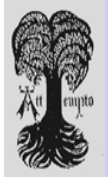
Probleme der „natürlichen“ Varianten

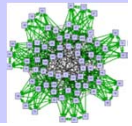
Nicht blockierendes Send

Buffer sprengen, keine Bestätigung

Blockierendes Receive

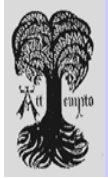
Verlorene Meldungen, Sender ausgefallen

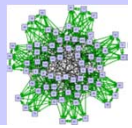




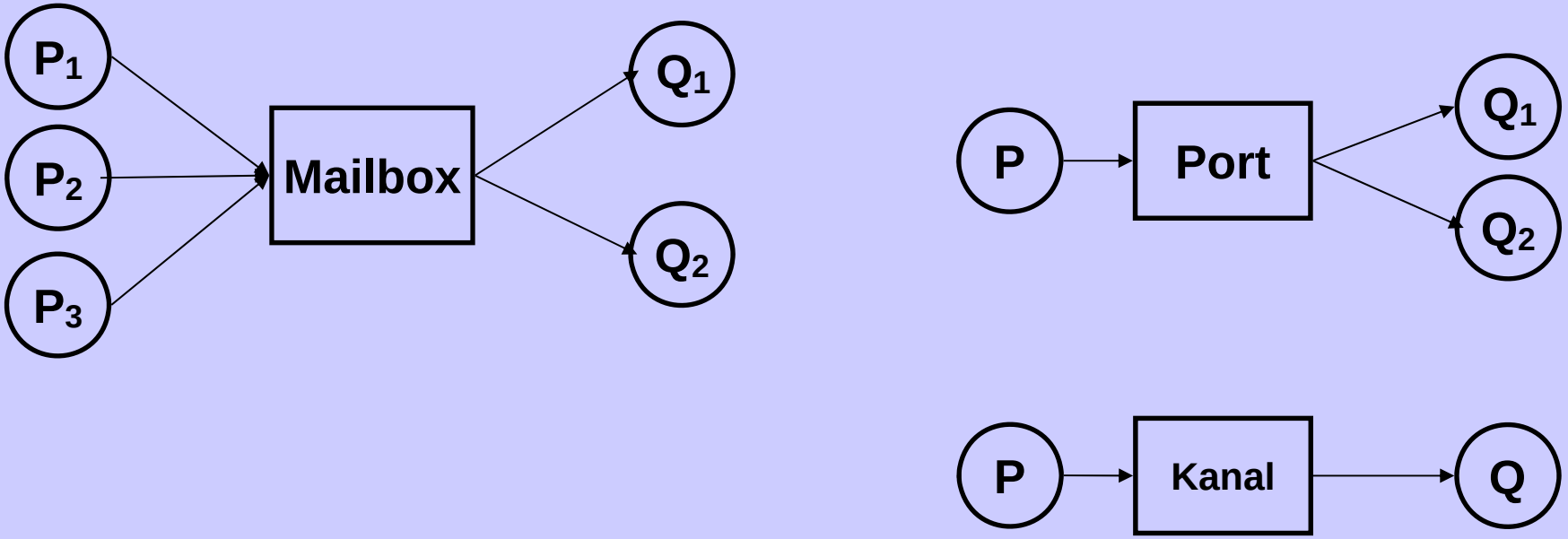
Adressierung

- Direkte Kommunikation
  - Explizite Angabe des Partnerprozesses
    - Automatische Erstellung des Kommunikationskanals
    - keine mehrfachen Kanäle zwischen gleichen Prozessen
- Indirekte Kommunikation
  - über Mailbox oder Port
    - Einrichten nötig
    - Mehrfache Mailboxes/Ports möglich

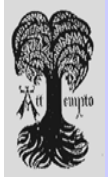


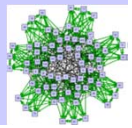


Adressierung



- Problem indirekter Kommunikation
  - Synchronisation nötig beim gleichzeitigen Schreiben oder Lesen mehrerer Prozesse.

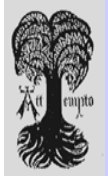


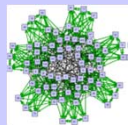


Message Passing – Producer/Consumer

```
mailbox_t fullbox, emptybox;



producer()
{   int item; message_t m;
    while (1)
    {   produce(&item);
        /* Receive empty envelope
              message */
        receive(emptybox, &m);
        /* Put item in a message
              envelope */
        build(&m, item);
        send(fullbox, &m);
    }
}
```

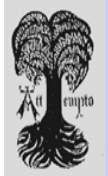
```
consumer()
{    /* Initialize: Create N empty
           envelopes */
    int item; message_t m;
    for (i = 0; i < N, i ++)
        send (emptybox, &m);

    while (1)
    {   receive(fullbox, &m);
        /* Get item from message */
        extract(&m, &item);
        /* Send envelope back */
        send(emptybox, &m);
        consume(item);
    }
}
```

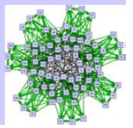




Monitore

- erfunden von Per Brinch Hansen und verbessert von Hoare
- Monitore sind ein höheres Programmiersprachenkonzept, das das Schreiben nebenläufiger Programme unterstützt.
- Datenobjekte und darauf agierende Prozeduren werden zusammengefasst ($\rightarrow$ objektorientiertes Programmieren)
- Compiler übersetzt derart, dass sich jeweils nur ein einziger Prozess im Monitor aufhalten kann.
- Vorteil:
  - Einzelne Datenobjekte müssen nicht mehr separat durch Locks geschützt werden.
- Gefahr:
  - Ineffizienz durch zu grobe Granularität, da eventuell nur kleine Teile der Monitorprozeduren wirklich exklusiv zu sein brauchen.

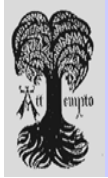


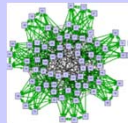


Monitore

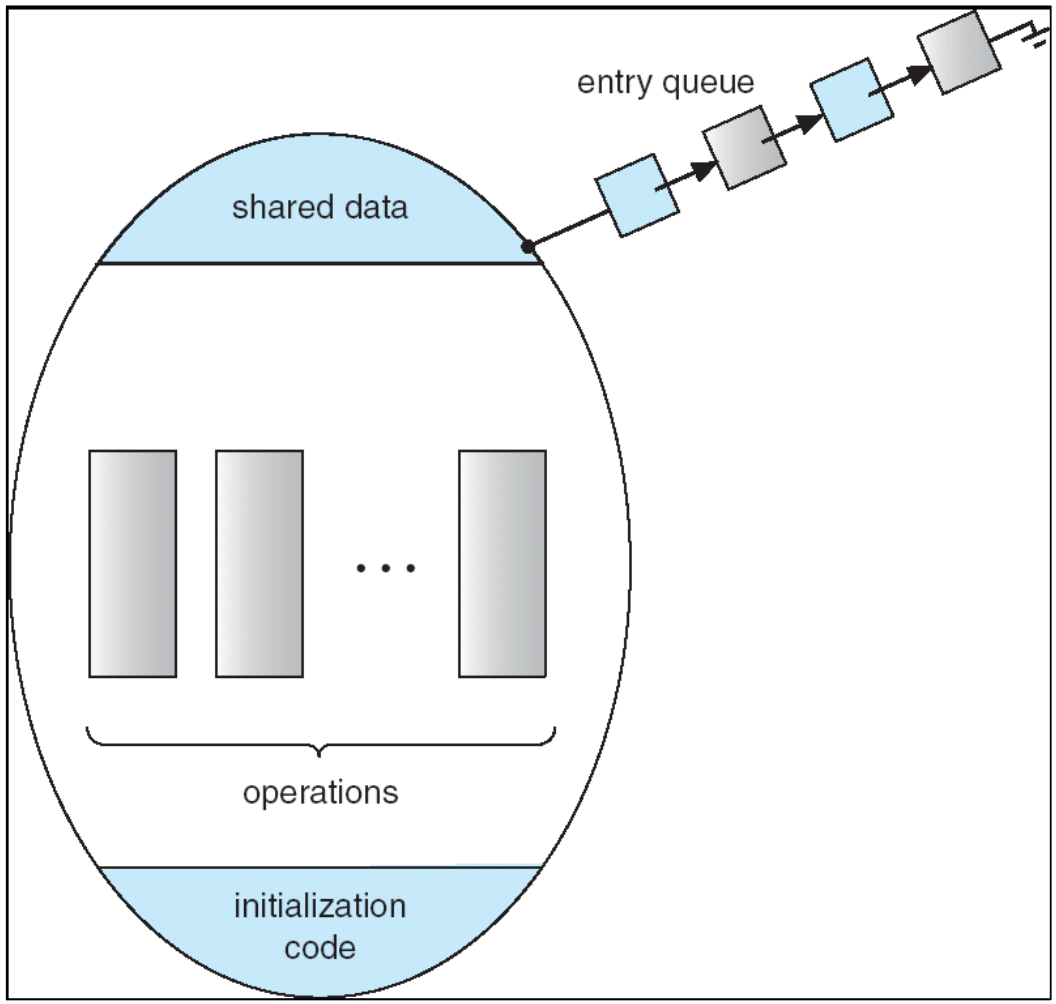
Beispiel: Zähler

```
monitor counter;
  export inc, dec, result;
  var count: integer;

  procedure inc
    begin  count:= count+1;   end;
  procedure dec
    begin  count:= count-1;   end;
  function result;
    begin  return count;   end;
begin
  count :=0;
end;
```

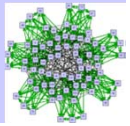




Monitore



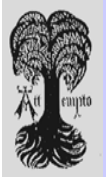
Silberschatz, Abb. 6.17

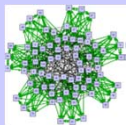


Monitore – Producer/Consumer

- Analog zu den Conditions für POSIX-Threads gibt es Conditions für Monitore
  - Geben implizit kritische Daten frei (und bekommen sie wieder)

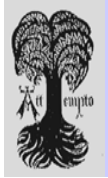
```
Monitor Buffer
{   condition nonfull, nonempty;
    buffer B[N]; int count = 0;

    void enter(x)
    int x;
    {   if (count == N) wait(nonfull);
        B[count] = x;
        count = count + 1;
        if (count == 1) signal(nonempty);
    }

    void remove(y)
    int* y;
    {   if (count ==0) wait(nonempty);
        *y = B[count];
        count = count - 1;
        if (count == N - 1) signal(nonfull);
    };
}
```

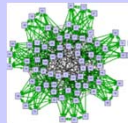




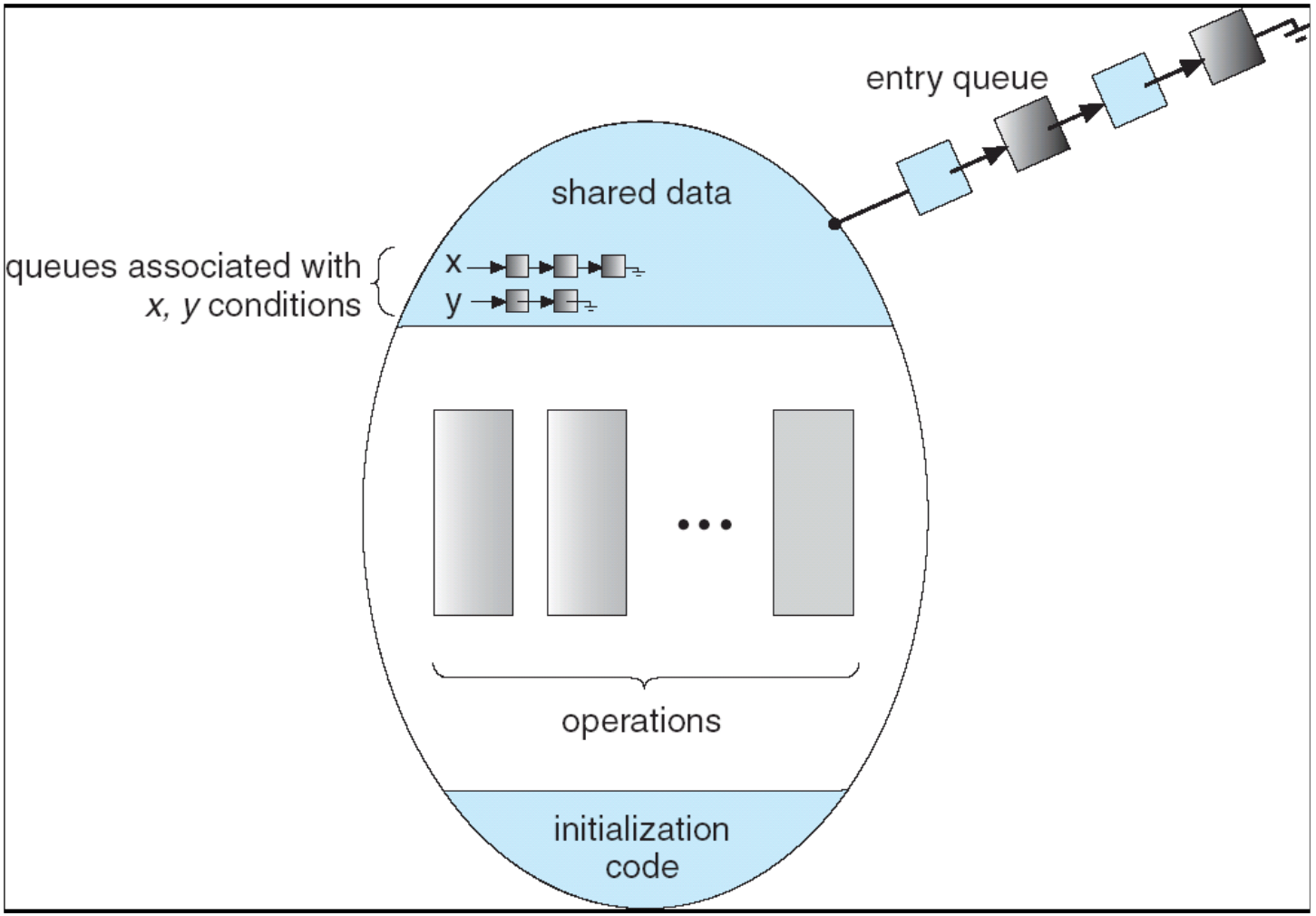
Monitore – Producer/Consumer

```
void producer()
{  item_t item;
   while (1) {  produce(& item);
                Buffer.enter(item);
             }
}

void consumer()
{  item_t item;
   while (1) {  Buffer.remove(& item);
                consume(item);
             }
}
```

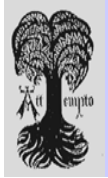


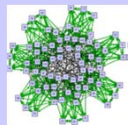


Monitore



Silberschatz, Abb. 6.18





Äquivalenz der Konzepte

- Alle betrachteten Konzepte
  - Messages
  - Monitors
  - Semaphores
  - mutex/conditions

  sind äquivalent.
- Überblick
  - Concurrent Euklid und –Pascal: Monitors
  - JAVA: Monitors (durch synchronized)
  - UNIX System V: Semaphore und Nachrichten
  - C Threads: mutex/conditions