S1

Donnerstag, 11. Dezember 2014 07:33

Prozesse und Threads

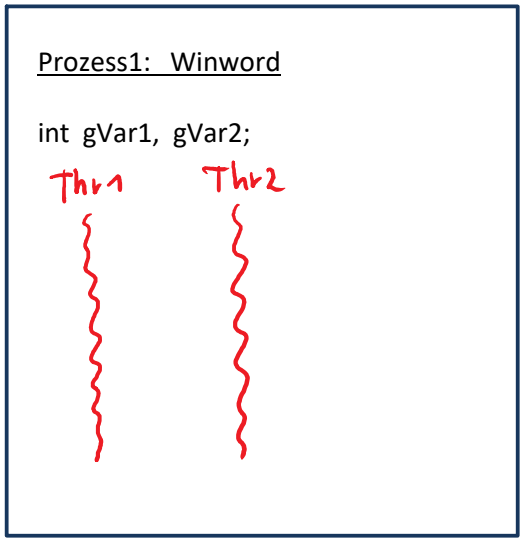
Concurrent

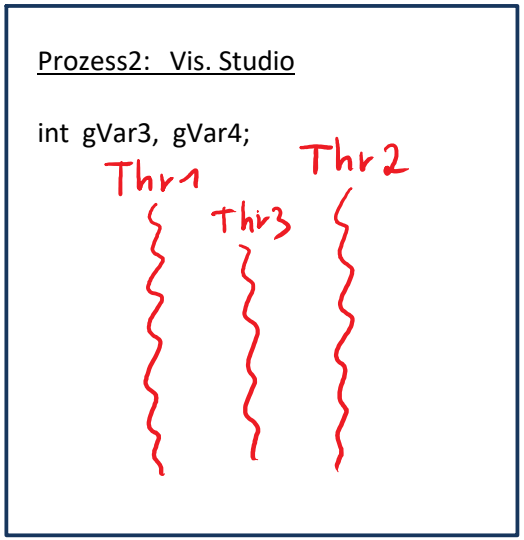
.NET

Task Parallel Library

Parallel For

transport





Die Threads innerhalb der Prozesse laufen quasiparalell (bei Multicore manchmal wirklich paralell) und haben unbeschränkten Zugriff auf die globalen Daten innerhalb ihres Prozesses.
Es gibt keine **einfache** Möglichkeit für Prozess1 auf Daten innerahlab von Prozess2 zuzugreifen.

Die Paralellität von 2 Threads in einem Prozess kann ich auch durch 2-Prozesse ohne Verwendung von Multithreading erreichen.

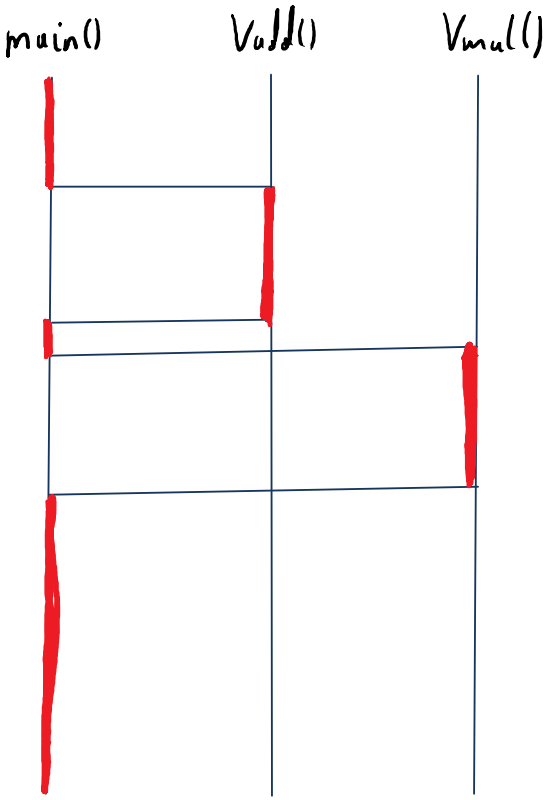
Beim Programmiermodell mit Prozessen ist es aus der Sicht der Kommunikation egal ob sich 2 Prozesse auf derselben Maschine befinden oder ob sie sich auf 2 Maschinen irgendwo im Internet befinden

Jede auf einem heutigen Betriebssystem aktive Applikation ist ein Prozess.

S2

Dienstag, 9. Dezember 2014 10:37

Sequenzieller Funktionsaufruf



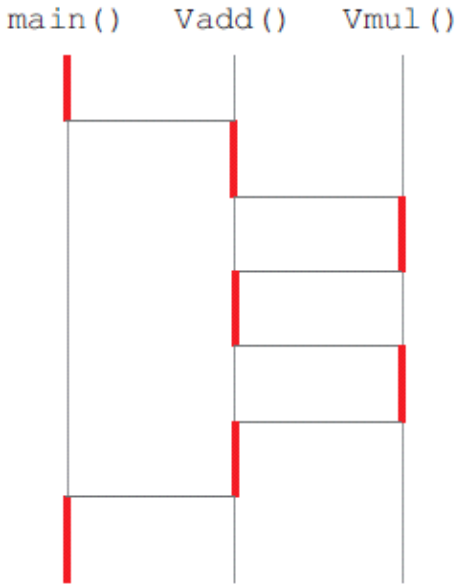

```
float A[1000], B[1000], C[1000], D[1000];

void main()
{
  Vadd(C,A,B);
  Vmul(D,A,B);
}

Vadd(C[],A[],B[])
{
   for(i=1; 1000)
     C[i]=A[i]+B[i];
}

Vmul fehlt noch !!!
```

Paralell mit 1Prozessor Maschine

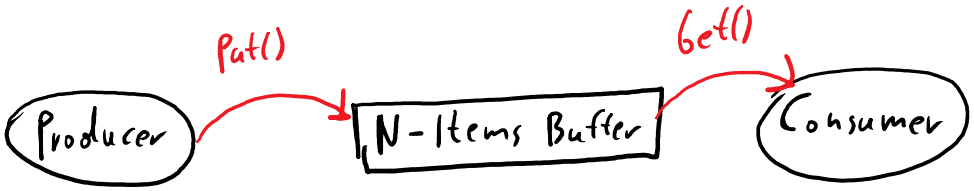
```
void main()
{
  t1=StartThread(Vadd(C,A,B));
  t2=StartThread(Vmul(D,A,B));

  // wartet bis t1 und t2 beendet sind
  join(t1,t2);
}
```

main kommt auch noch dran !!!

Zeitablauf für Multicore



Das Erzeuger Verbraucher Problem



Producer und Consumer erzeugen im Mittel die gleiche Datenrate.
Es gibt jedoch Datenraten-Peaks beim Producer die vom Buffer aufgefangen werden müssen.
Bei einem Put() auf einen vollen Puffer muß der Producer blokiert werden (schlafen),
und vom Consumer wieder aufgeweckt werden wenn wieder Platz im Puffer ist.
Dasselbe gilt beim Consumer für ein Get() auf einen leeren Puffer.

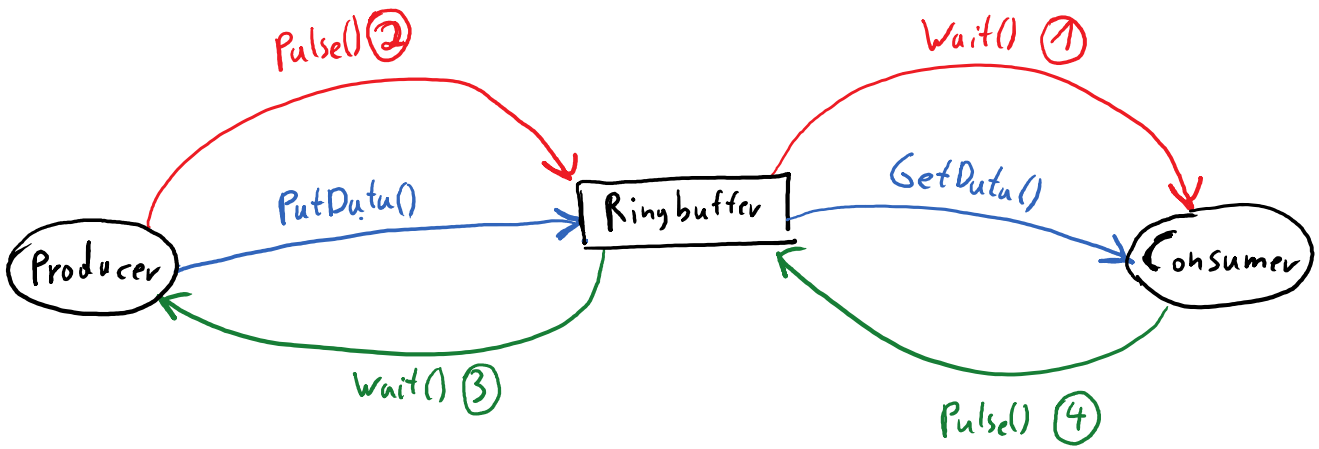
Der Buffer als Datenstruktur kann auf verschiedenste Weise implementiert sein
(z.B RingBuffer oder verkettete List).
Die Implementierung der Datenstruktur hat nichts mit der Synchronisationsaufgabe zu tun
das sind 2 verschiedene Dinge.

Betriebssystem Klassiker:

Andrew S. Tannenbaum (Minix Unix Lernsystem)

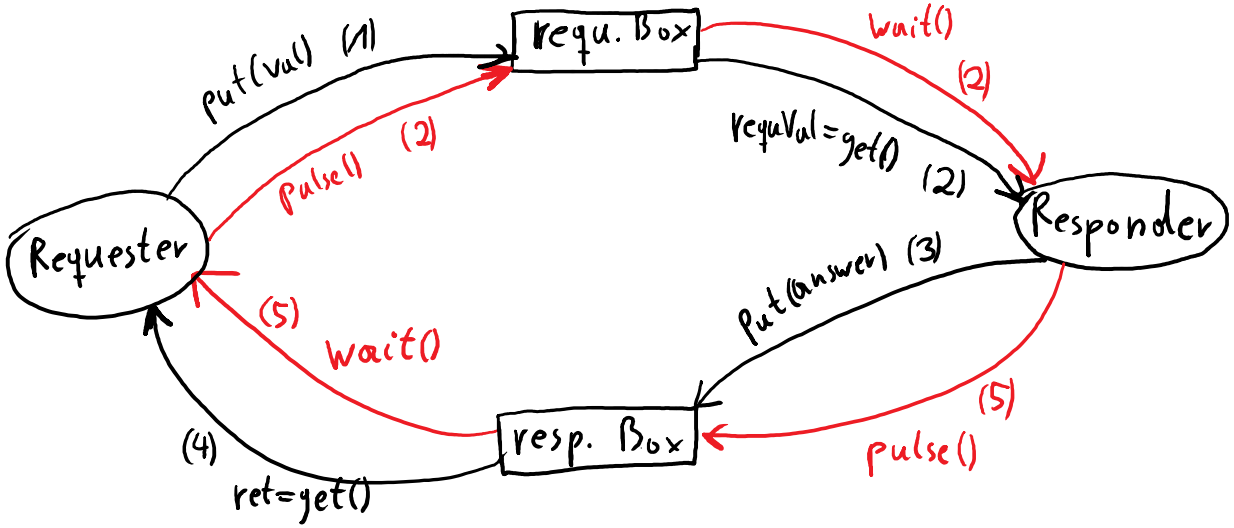
ProdCons.cs

Dienstag, 9. Dezember 2014 10:37



Wenn der Puffer weder ganz leer noch ganz voll ist
werden vom Producer und Consumer nur PutData() und GetData() aufgerufen.
Wird der Puffer einmal ganz leer kommt es zum Pulse() Wait() Szenario 1,2
Wird der Puffer einmal ganz voll kommt es zum Pulse() Wait() Szenario 3,4

RequestResponse.cs




```
Hit Enter to finish.
Request:1
Got Request:1
Send Answer:3
Answer:3

Request:2
Got Request:2
Send Answer:4
Answer:4

Request:3
Got Request:3
Send Answer:5
Answer:5

Request:4
Got Request:4
Send Answer:6
Answer:6

Request:5
Got Request:5
Send Answer:7
Answer:7
```

Ereignis Variablen

```
int   cnt;

ThrA()
{
    while(true) {
    // cnt für die Dauer der Änderung sperren
    lock(cnt)
    {
        cnt++
        // ThrB benachrichtigen, daß sich cnt geändert hat
        Pulse(cnt);|
    }
    }
}

// Wartet bis cnt==5 ist
// Angabe: ThrB() fragt cnt ständig ab und meldet sich wenn
// cnt==5 ist
ThrB()
{
    while(true)
    {
        Enter(cnt);
        if( cnt==5 ) {
            printf("5 erreicht);
            Exit(cnt); // Mutex freigeben
        }
        else
            Wait(cnt);
            // schlafen bis ich an cnt etwas geändert hat
    }
}
```

Wait macht ein implizites Exit()

Ereignis Variablen ermöglich das folgende Szenario welches beim Concurent Programming immer wieder vorkommt:

ThrB() wartet in einer while(true) Schleife bis eine Datenstruktur (in diesem Fall cnt==5) einen bestimmten Wert hat.
Um **BusyWaiting zu vermeiden** überprüft ThrB() die Bedingung und suspendiert sich selbst mit wait() wenn die Bedingung nicht erfüllt ist.

Bei jeder Änderung der Datenstruktur wird ThrB() durch ThrA() mit Pulse() afgeweckt damit ThrB() die Bedingung wieder neu evaluieren kann.

Während **cnt geändert wird** haben sowohl ThraA() als auch ThrB() durch lock() exklusieven Zugriff auf cnt.

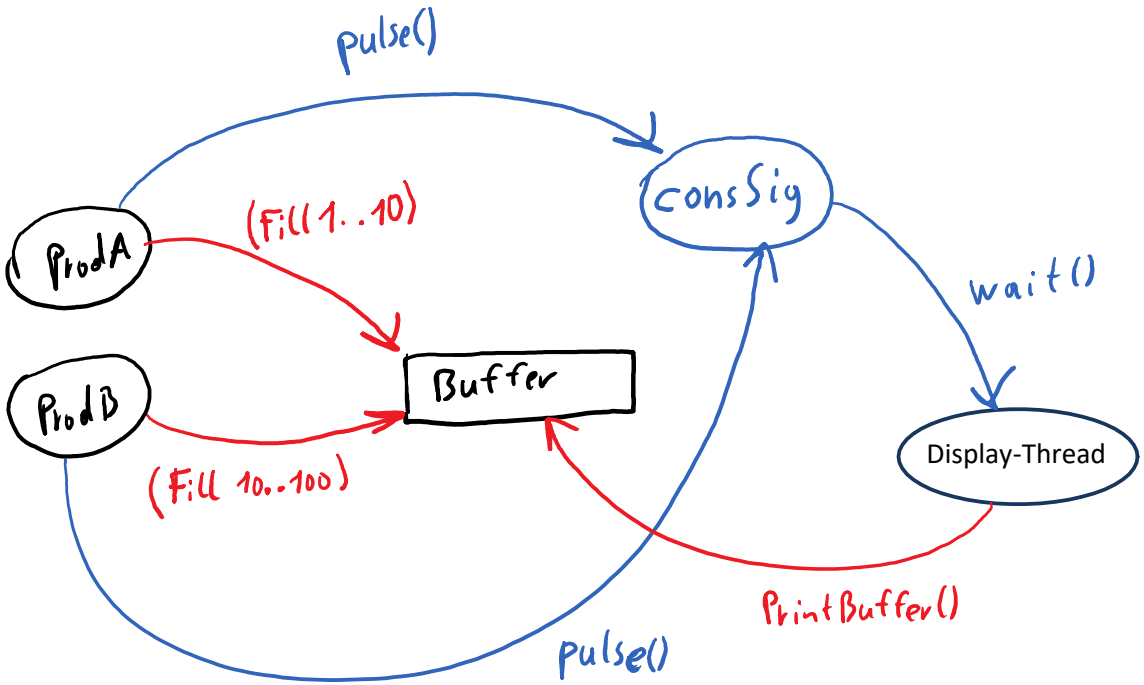
Dh. im Ereigniss Variablen Szenario kommen
gegenseitiger Ausschluss und
einseitige Benachrichtigung vor

Im Producer Consumer Beispiel kommen Ereignis Variablen in den Full/Empty Fällen vor.

Ereigniss Variablen bestehen typischerweise aus einer Datenstruktur und einem Synchronisationsobjekt.
Im hier gezeigten Bsp. wird cnt als Datenstruktur und Synchronisationsobjekt verwendet.

TwoProducers.cs

Donnerstag, 8. Januar 2015 08:25



ProdA und ProdB befüllen den Puffer
1x synchronisiert und 1x unsynchronisiert .
Jedesmal wenn ProdA oder ProdB mit dem Befüllen fertig sind
wird der Display-Thread über consSig benachrichtigt und gibt
den Inhalt des Puffers auf der Konsole aus.
Je nachdem ob der Puffer synchronisiert oder unsynchronisiert
befüllt wurde sind die angezeigten Daten vermischt oder konsistent

S7

Donnerstag, 8. Januar 2015 08:25

Darf jeder Thread auf die GUI schreiben ?
Oder anders gefragt darf der Worker1 im FormsAndThreads-Bsp. den folgenden Aufruf ausführen:

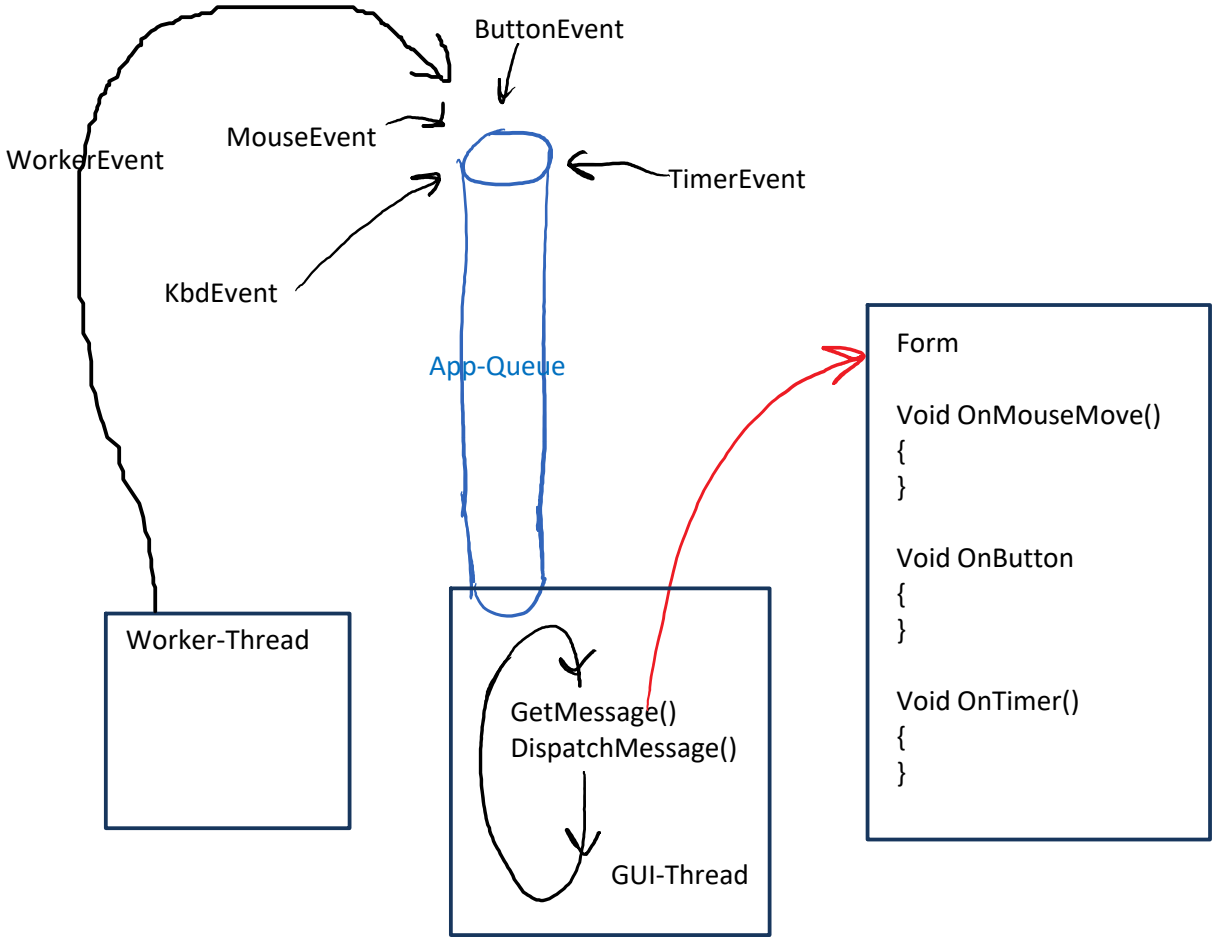
```
frm.TestBox1.Text="Hallo"
```

Antwort: Nein das darf der Worker1 nicht da nur der GUI-Thread Anzeigedaten in der Form ändern darf.

Es gibt 2 Lösungen für das Problem:

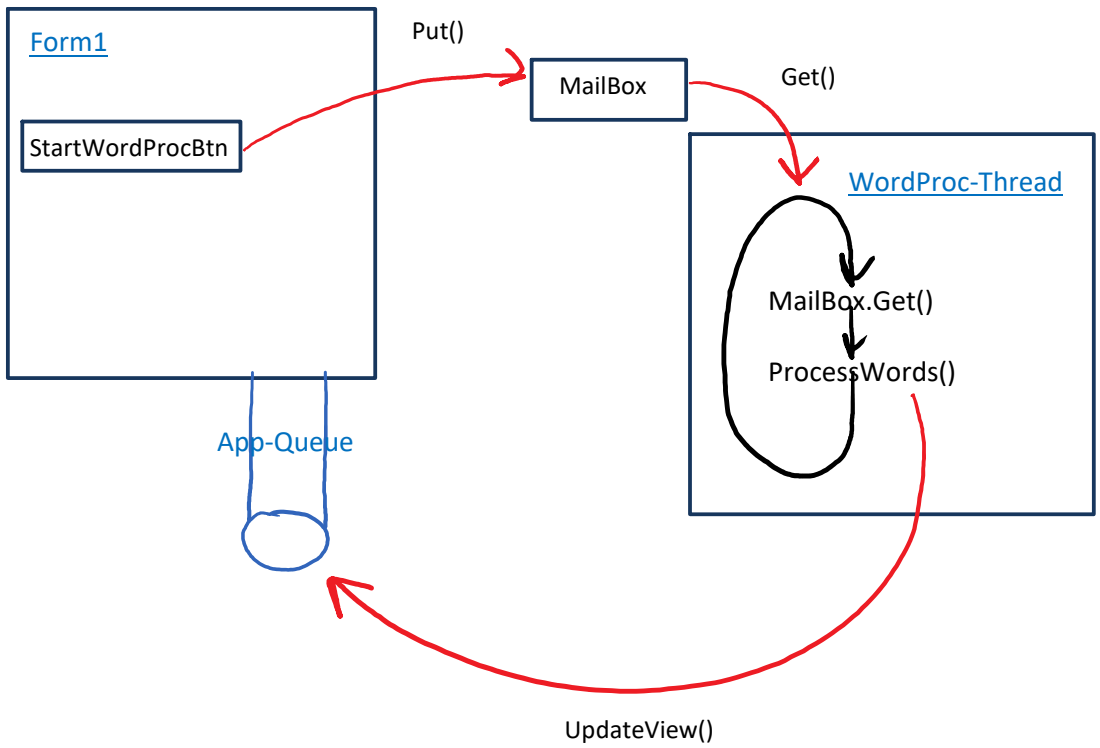
1. Daten des Workers aus der GUI mit einem Timer abfragen
2. Messages vom Worker an den GUI-Thread in die App-Queue stellen (putten).

Jedem Prozess ist in Windows eine App-Queue zugeordnet.
In diese App-Queue werden von Windows GUI-Ereignisse wie z.B. ButtonEvent, MouseEvent, TimerEvent eingequeued.
Der GUI-Thread liest diese Ereignisse aus der App-Queue aus und ruft die dazu passenden Methoden der Form auf.
Die Form und auf der Form sichtbare Elemente (Controls) dürfen nur vom GUI-Thread verändert werden.

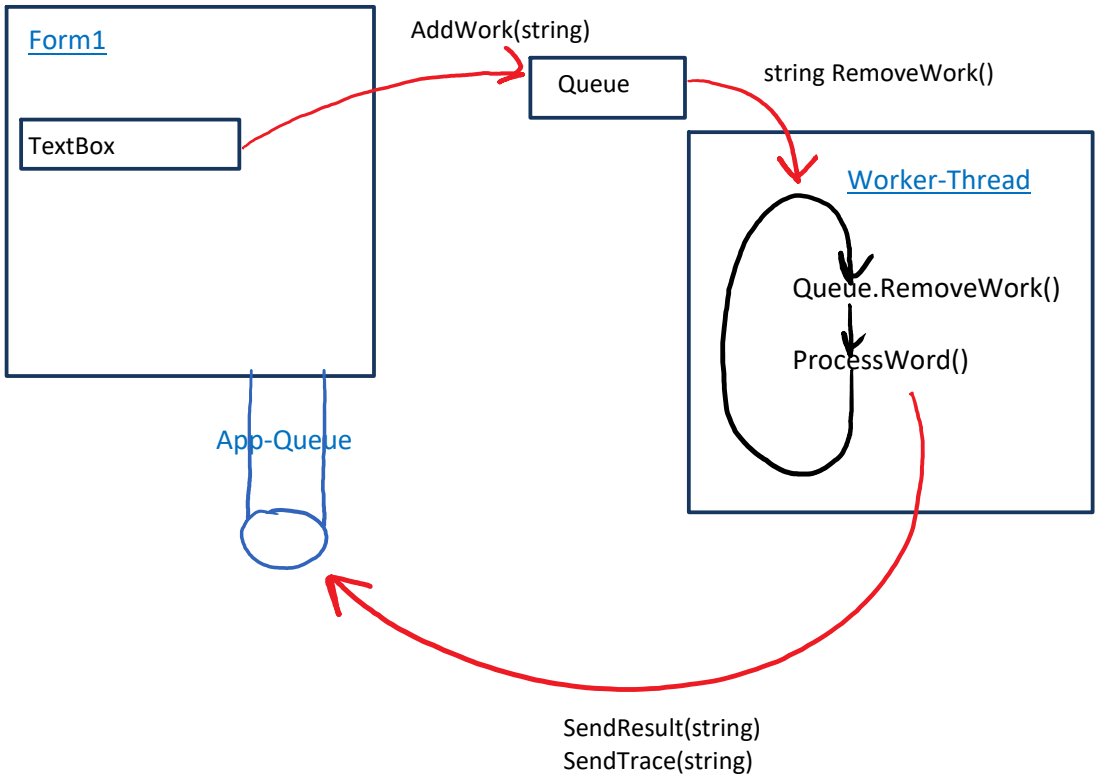


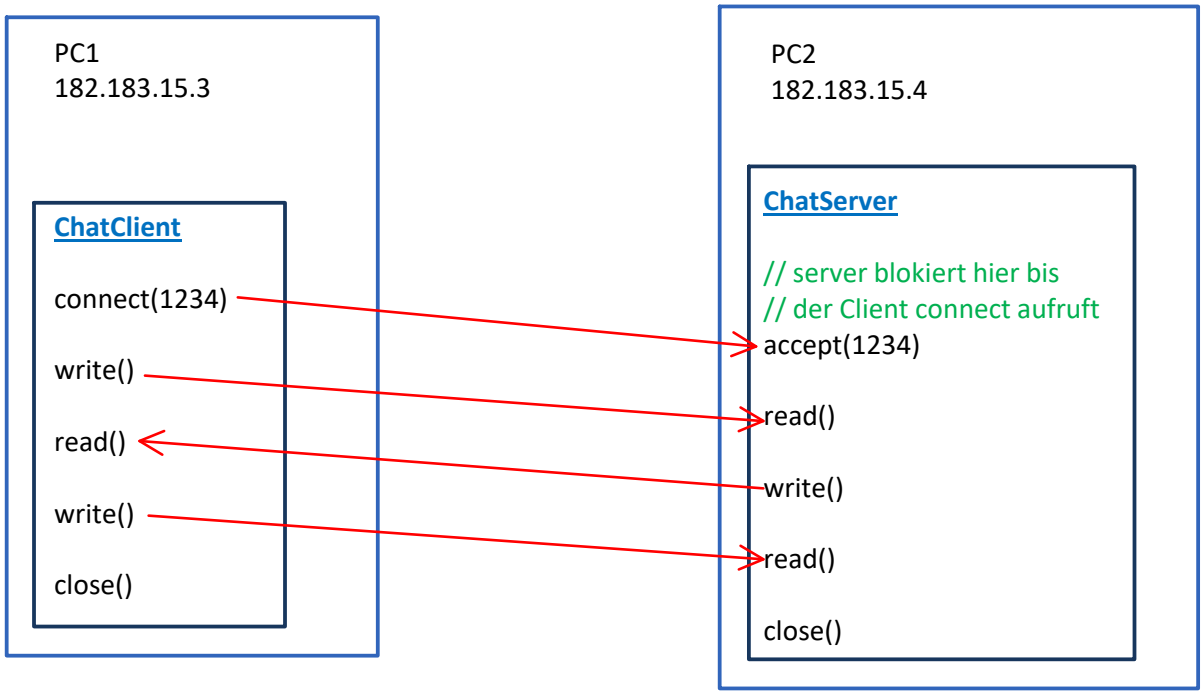
FormsAndThreads.cs

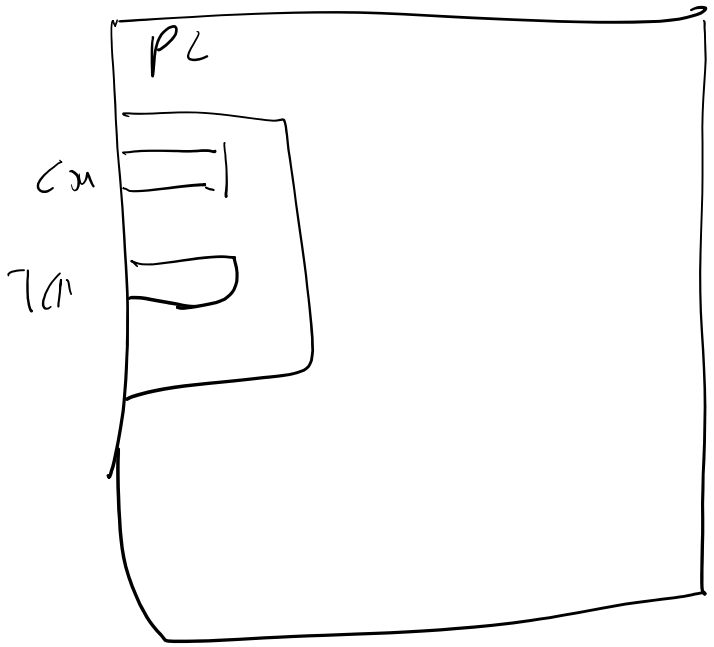
```
namespace WordProc
{
  public delegate void FuncType1(int aData);

  public partial class Form1 : Form
  {
    public static Form1 frm;
    FuncType1 WorkerMessageFuncPtr;
    WordProc wrp;


WorkerMessageFuncPtr = this.RecvMessageFromWorker;
frm = this;


 // wird im Context des GUI-Threads aufgerufen
 public void RecvMessageFromWorker(int aData)
 {
   _txtBox2.Text = wrp.outTxt.ToString();
   if (aData == 1)
     wrp.mbx.Put(1);
 }

// wird im Context des WorkerThreads aufgerufen
public void SendMessage2Form(int aData)
{
  this.BeginInvoke(WorkerMessageFuncPtr, aData);
}
```



StringProcessor.cs

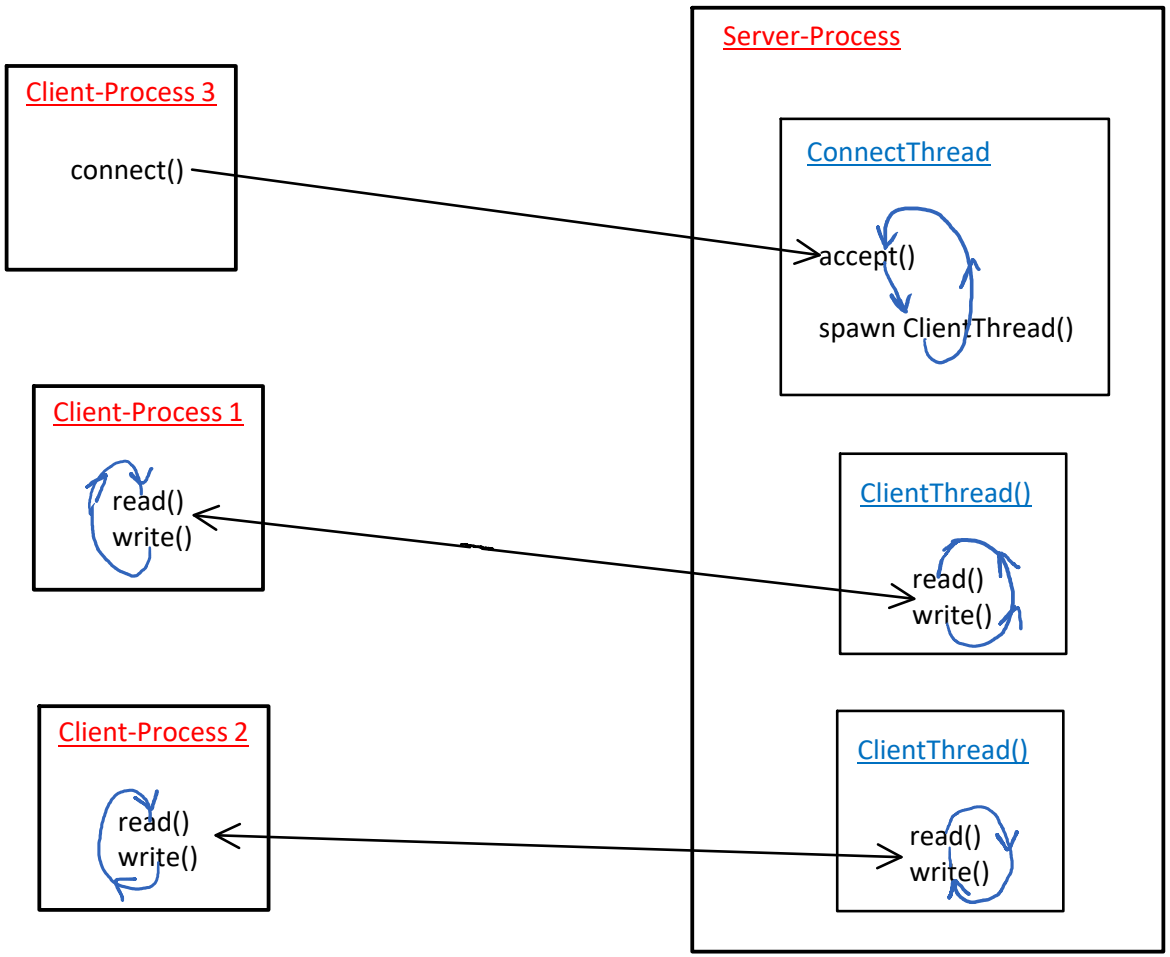


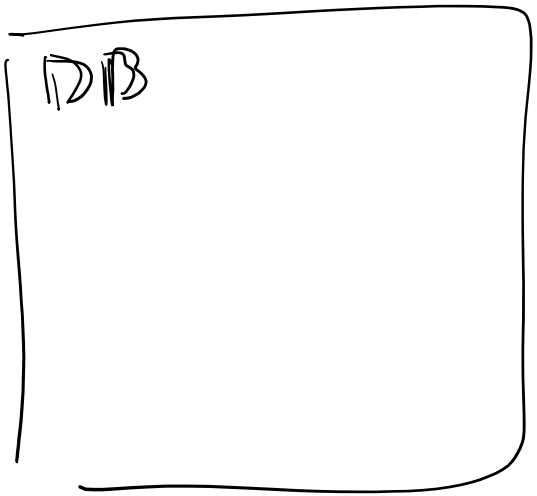




TCP/IP verbindet Prozesse auf der
gleichen Maschine oder auf einer anderen Maschine
im LAN oder auch WWW miteinander

Der *ConnectThread* ruft *accept()* auf und und blockiert an dem SystemCall.
Wenn sich ein *ClientProzess* mit *connect()* angemeldet oder anders gesagt mit *connect()* eine Verbindung aufbaut erzeugt der *ConnectThread* für die neue Verbindung einen *ClientThread* der mit dem *ClientProzess* die read/write Transaktionen abwickelt.
Beendet der *ClientProzess* die Kommunikation mit dem Server so muss auch *ClientThread* beendet und deleted werden.

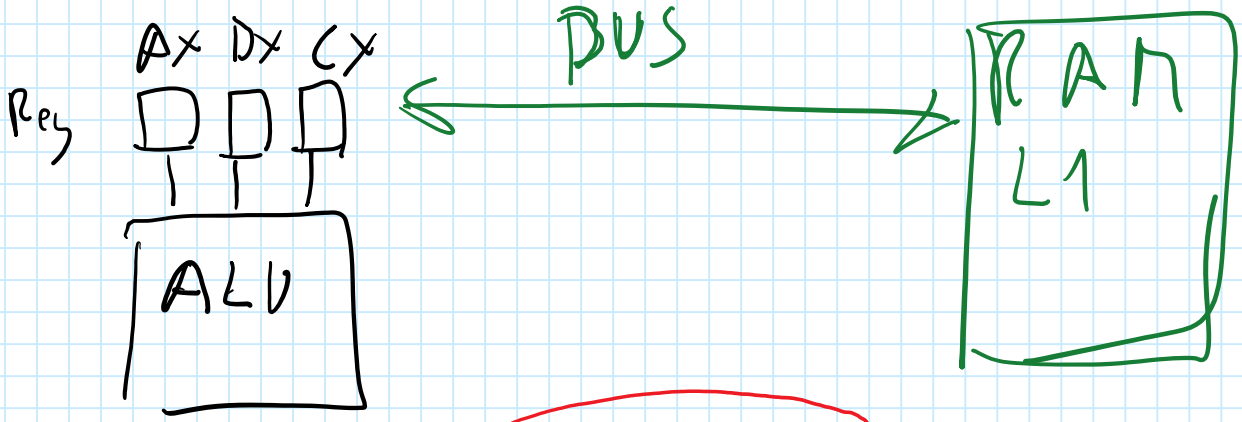




A [A1 A2 A3 A4]

B [B1 B2 B3 B4]

→ [A1 B1 A2 A3 B2 A4 B3 B4 ...]



ATOMAR

ADD AX, BX; AX = AX + BX;

ADD AX, 1; AX = AX + 1;

~ int yA;

```
Thrv()
{
    yA++;
    yA++;
    yA--;
}
```

Tannenbaum