Parallele Programmierung

Dipl.-Ing. Msc. Paul Panhofer Bsc.

12. Dezember 2022

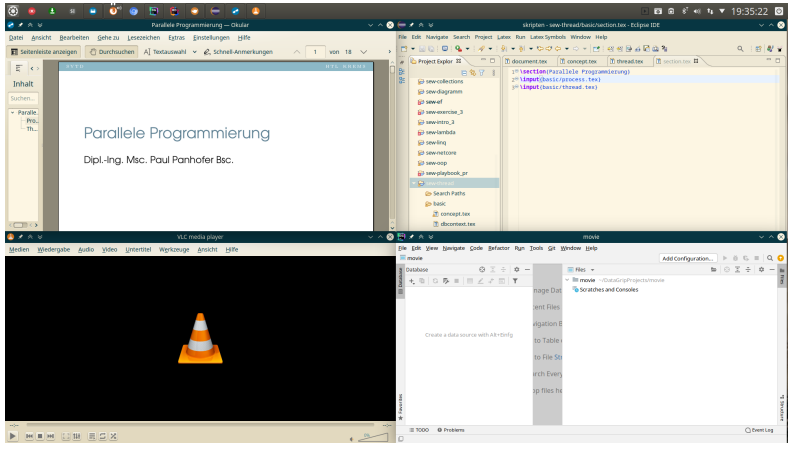
Prozess

Ein Prozess wird als ein in Ausführung befindliches **Programm** bezeichnet.

Betriebssysteme sind für die Verwaltung von Prozessen verantwortlich.

Prozess

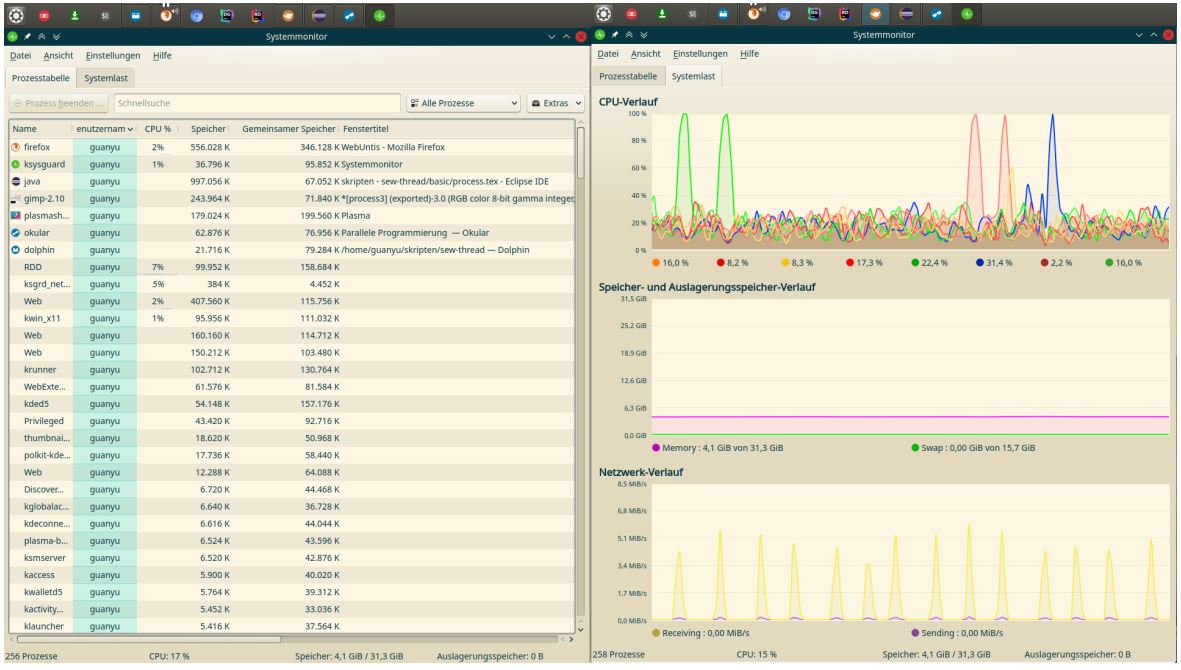
Programmausführung



z.B.: *ps, top*

Prozess

Prozessverwaltung



Prozess

Prozesswechsel

Prozesse benötigen in Folge ihrer Ausführung Ressourcen.

Ein Prozess wechselt dabei zwischen der **Nutzung** einer CPU und dem **Warten** auf eine Ein- bzw Ausgabe.

Prozess

Verwaltung von Prozessen

Das Betriebssystem verwaltet Prozesse in Prozesstabellen. Prozesstabellen enthalten die zur **Verwaltung** von Prozessen notwendinge Information.

Die Einträge einer Prozesstabelle werden als Prozesskontrollblock bezeichnet. Ein einzelner Prozesskontrollblock wird zur Verwaltung eines **Prozesses** verwendet.

Prozess

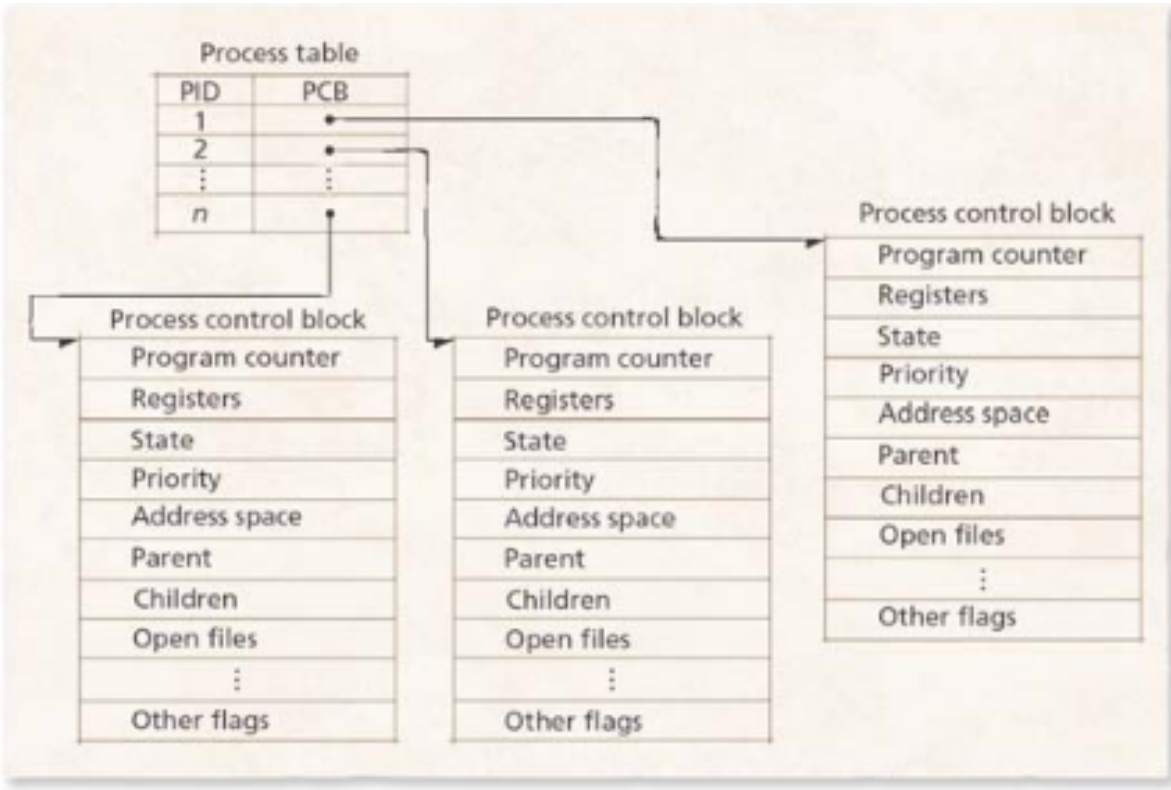
Prozesskontrollblock

Inhalt eines **Prozesskontrollblocks**:

- Vom Prozess angeforderte Ressourcen (z.B.: Dateien)
- Die dem Prozess zugeordneten Bereiche des Hauptspeichers
- Prozessidentifikation (PID)
- Prozesszustand
- Inhalt der Prozessregister

Prozess

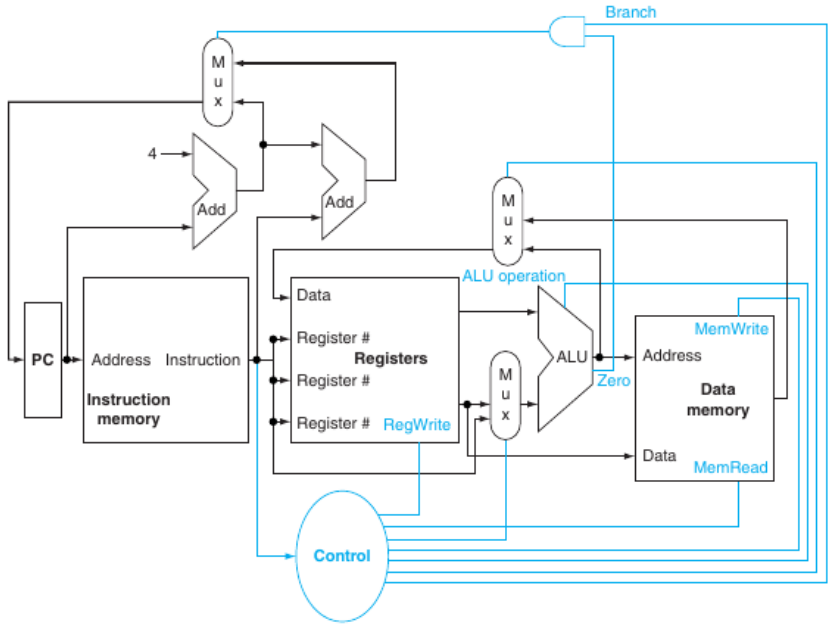
Prozesskontrollblock



Prozess

Prozessausführung

Ein Prozess wird vom Betriebssystem **ausgeführt**, indem er in die Register einer CPU geladen wird.



Prozessausführung

Ein Prozess befindet sich in **Ausführung**, wenn ihm eine CPU zur Ausführung zugeordnet ist.

Prozesse werden dazu in den Zustand Running versetzt.

Prozessausführung

Prozesszustände

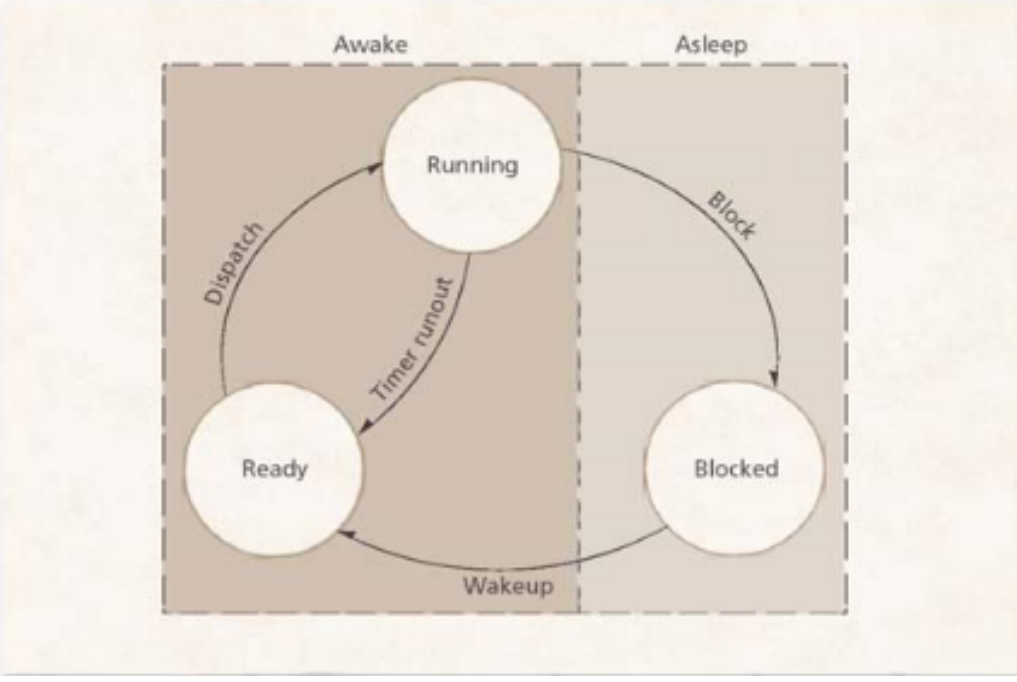
Prozesse werden gesteuert indem ihr **Zustand** verändert wird.

Generell werden für Prozesse 3 Zustände abstrahiert:

- Running
- Ready
- Blocked

Prozessausführung

Prozesszustände



Prozessausführung

Prozesszustände

- Running: Der Prozess befindet sich in **Ausführung**.
- Ready: Der Prozess **wartet** bis ihm das Betriebssystem die CPU zur Ausführung zuordnet.
- Blocked: Der Prozess benötigt zur Ausführung eine **externe Ressource**.

  Bis die Ressource vom Betriebssystem geladen werden kann, wird der Prozess in den Zustand Blocked versetzt.

Prozessausführung

Parallel vs. Concurrent

Für die Ausführung von Prozessen werden 2 **Formen** unterschieden: Parallel vs. Concurrent.

Prozessausführung

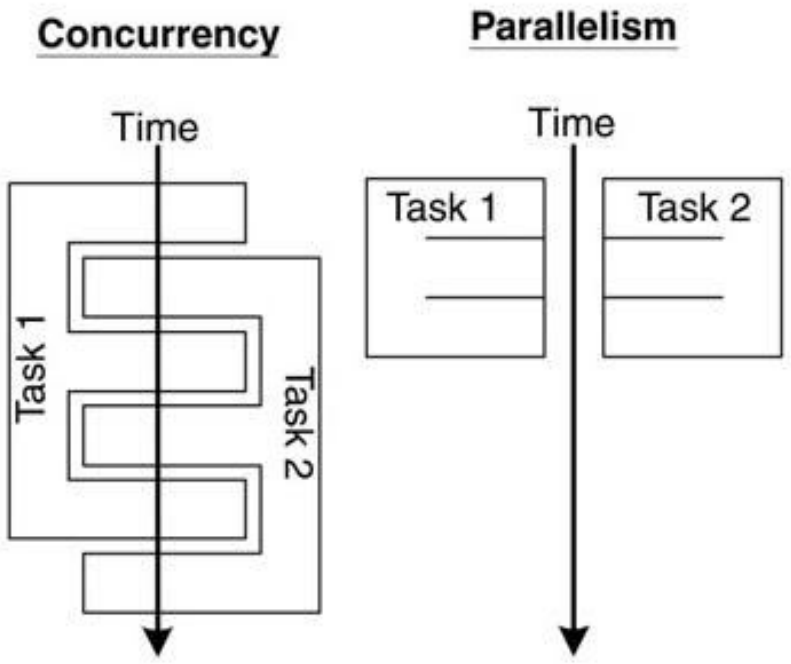
Parallel vs. Concurrent

A system is said to be concurrent if it can support two or more actions **in progress** at the same time.

A system is said to be parallel if it can support two or more actions executing **simultaneously**.

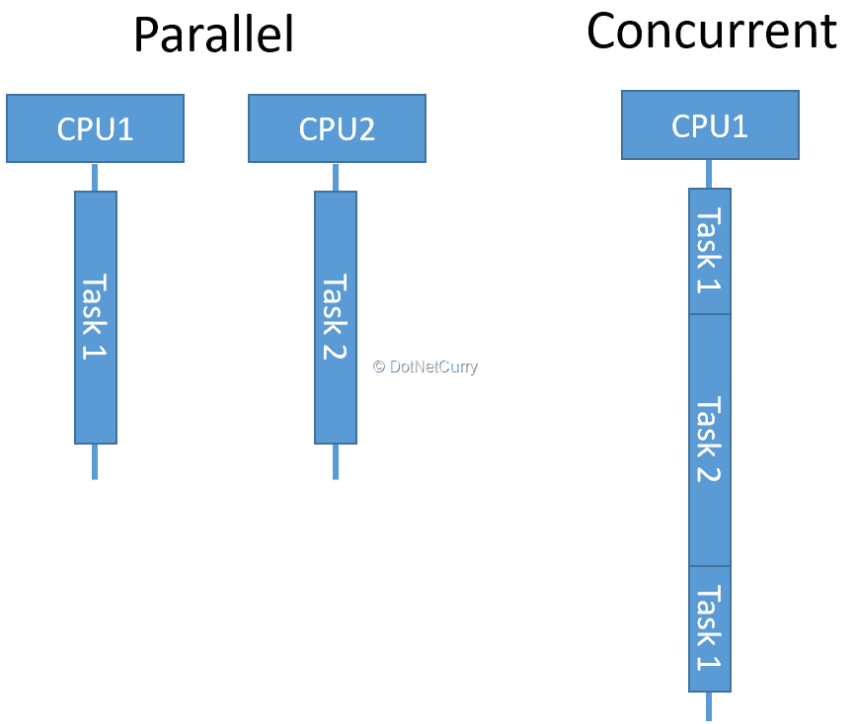
Prozessausführung

Parallel vs. Concurrent



Prozessausführung

Parallel vs. Concurrent



Prozessausführung

Concurrent

Bei der **nebenläufigen** Ausführung von Prozessen, wird den Prozessen abwechselnd die CPU zur Ausführung zugeteilt.

Soll eine neuer Prozess ausgeführt werden, werden die Daten des zuvor geladenen Prozesses in den Speicher geschrieben und der Inhalt des aktuellen Prozesses in die Register der CPU geladen.

Prozessausführung

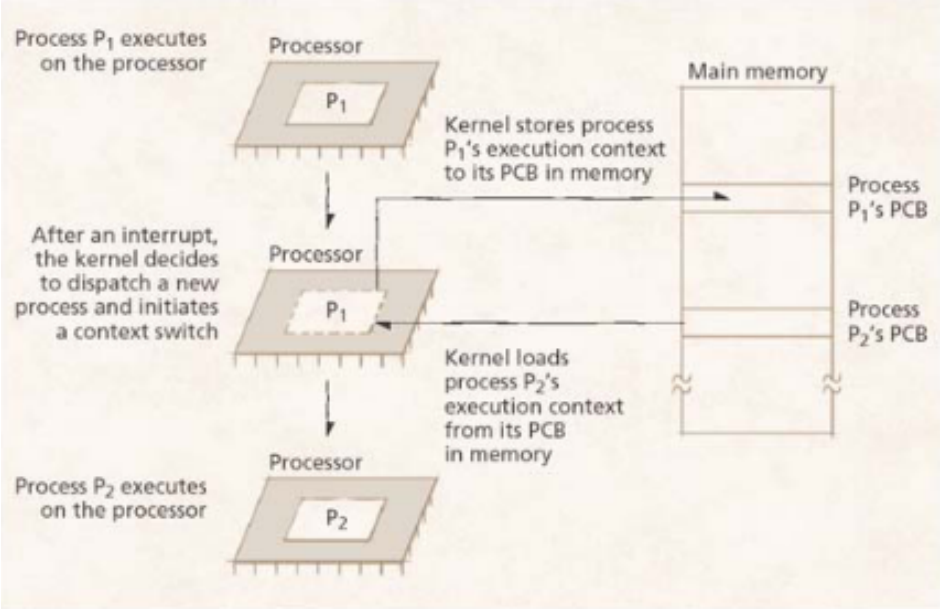
Concurrent

Das Laden eines neuen Prozesses wird als Kontextswitch bezeichnet.

Der Kontextswitch stellt auf **Betriebssystemebene** eine zeititensive Operation dar.

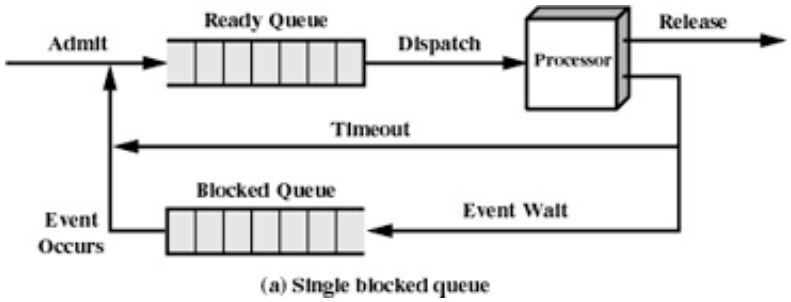
Prozessausführung

Kontextswitch

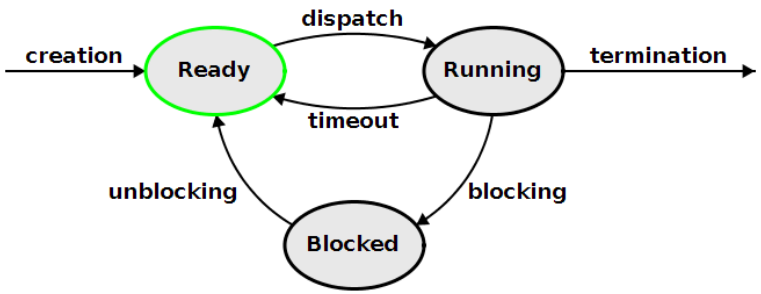


Prozessausführung

Kontextswitch



(a) Single blocked queue



Prozessausführung

Kontextswitch

Ein Kontextswitch beschreibt die **Abfolge** folgender Schritte:

- Aktualisierung und **Sicherung** des aktuellen Kontroll-blocks
- **Auswahl** eines anderen Prozesses zur Ausführung
- Wiederherstellung und Aktualisierung des Kontroll-blocks des neuen Prozesses

Prozessausführung

Kontextswitch: Motivation

Prozesse benötigen in Folge ihrer Ausführung **Ressourcen**.

Ein Prozess wechselt damit zwischen der **Nutzung** einer CPU und dem **Warten** auf eine Ein- bzw Ausgabe oder ein anderes Ereignis.

Thread

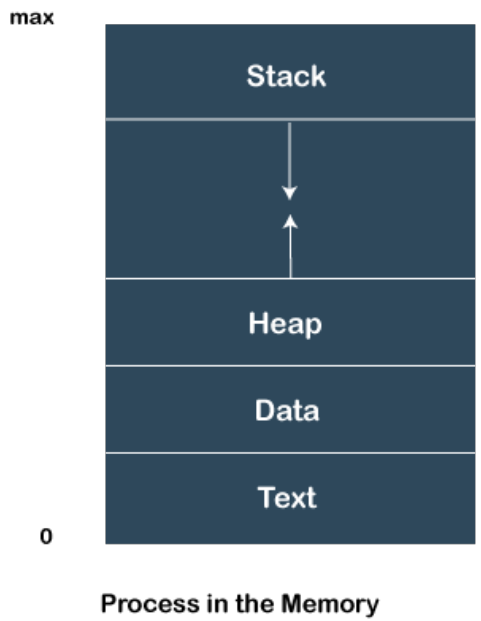
Prozess

Als Prozess wird ein in **Ausführung** befindliches Programm bezeichnet.

Thread

Prozess

Das Betriebssystem weist einem Prozess einen Teil des Speichers zu.



Process in the Memory

Thread

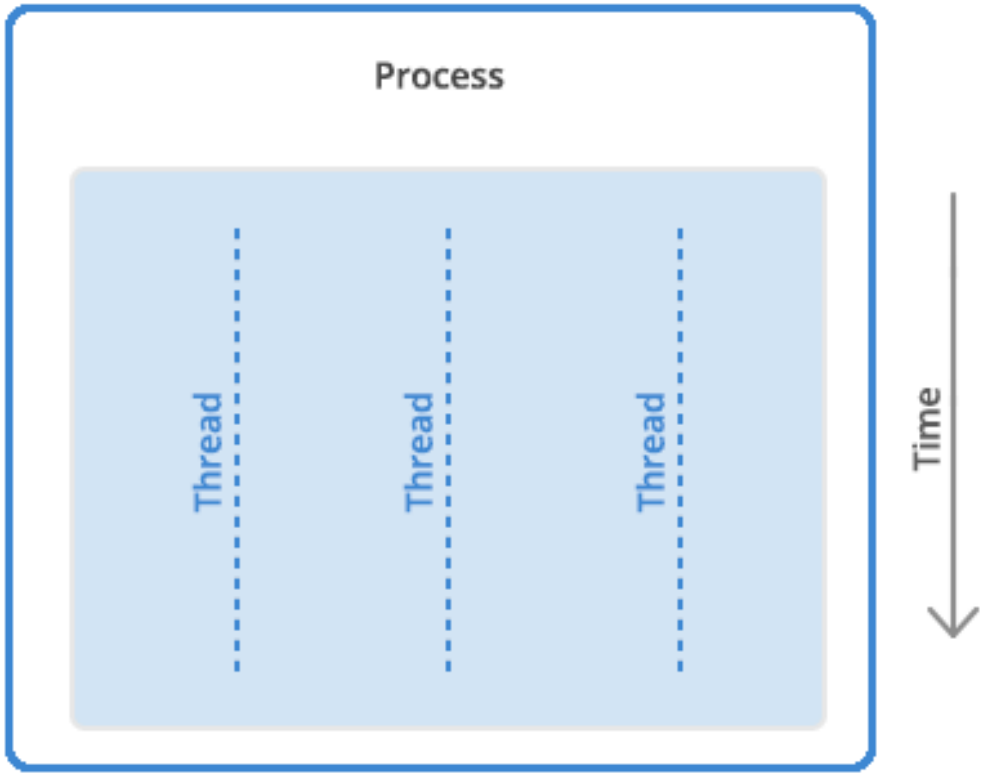
Prozess vs. Thread

Ein Thread ist eine autonome Einheit innerhalb eines Prozesses zur Ausführung von **Aufgaben**.

Konzeptionell ermöglichen Threads die nebenläufige Ausführung von Aufgaben innerhalb eines einzelnen Prozesses.

Thread

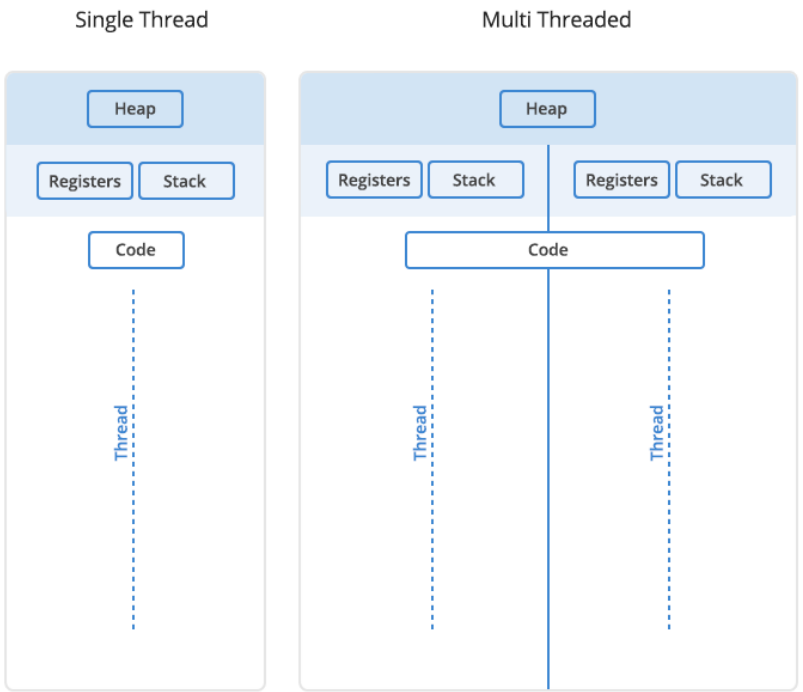
Prozess vs. Thread



Thread

Prozess vs. Thread

Threads teilen sich innerhalb eines Prozesses Teile seiner **Ressourcen** - (Heap).



Thread

Prozess vs. Thread

PROCESS	THREAD
Processes are heavyweight operations.	Threads are lighter weight operations.
Each process has its own memory space.	Threads use the memory of the process they belong to.
Inter-process communication is slow as processes have different memory addresses.	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to.
Context switching between processes is more expensive.	Context switching between threads of the same process is less expensive<./td>
Processes don't share memory with other processes.	Threads share memory with other threads of the same process.

TPL - Task Programming Library

Threads

Zur Ausführung nebenläufiger **Aufgaben** in einem Prozess, werden Threads verwendet.

Die Zahl von Threads, die ein Prozess verwalten kann, ist begrenzt.

TPL - Task Programming Library

Threads

Threads sind Betriebssystemressourcen. Zur Verwaltung von Threads ist kostspielig.

Threads als Ressourcen sind für die Anwendungsentwicklung zu **lowlevel**.

TPL - Task Programming Library

Task Programming Library

`.net Core` bietet mit der **TPL** - Task Programming Library - eine Library zur einfachen und effektiven Programmierung nebenläufiger Abläufe.

Die Task ist dabei der grundlegende **Baustein** der TPL.

TPL - Task Programming Library

Task

Eine Task abstrahiert einen **Request**. Im Gegensatz zu Threads können in einem Programm eine beliebige Zahl von Tasks gestartet werden.

TPL - Task Programming Library

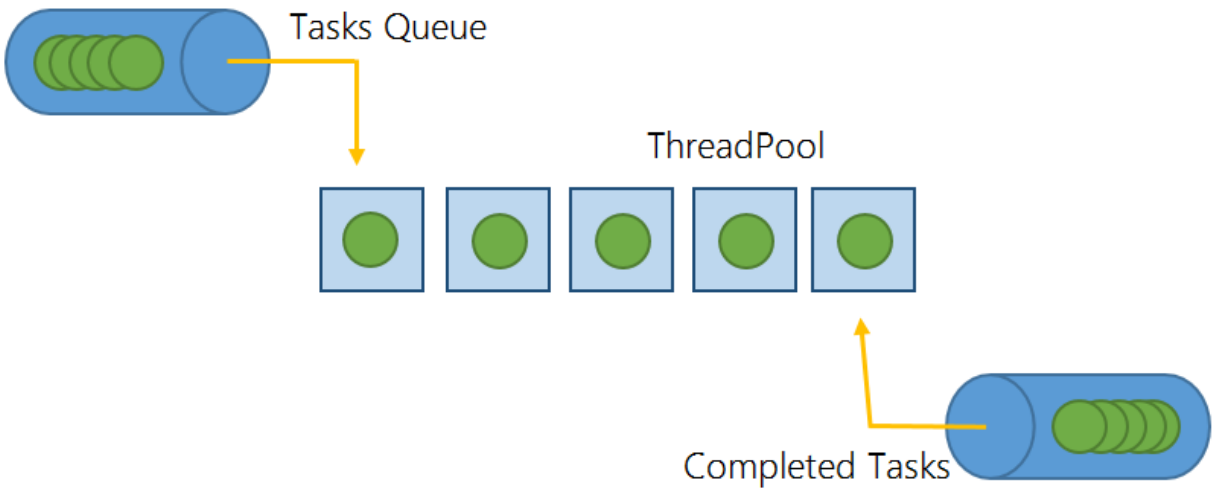
Task - Threadpool

Die TPL führt Tasks in Threads aus. Dazu verwaltet die TPL einen **Threadpool**.

Sobald ein Task ausgeführt wurde, wird der Thread zur späteren Verwendung in den Threadpool zurückgelegt.

TPL - Task Programming Library

Task



TPL - Task Programming Library

Task erzeugen

```
public static void Main(String[] args){
    // Der Task Constructor erwartet als Parameter
    // ein Delegate
    var task = new Task(
        () => {
            Console.WriteLine("alora .. ");
        }
    );

    // Ausfuehren der Task in einem Thread
    task.Start();
}
```

TPL - Task Programming Library

Task erzeugen

```
public static void Main(String[] args){
    // Anlegen und Ausfuehren einer Task
    var task = Task.Run(
        () => {
            Console.WriteLine("alora .. ");
        }
    );
}
```

TPL - Task Programming Library

Taskmethode: Wait

Tasks werden von der TPL in Backgroundthreads ausgeführt. Der Main Thread der Anwendung wartet damit nicht auf das Ergebnis einer Task.

Hinweis: Durch das Aufrufen der `Wait` Methode kann der main Thread solange **blockiert** werden bis ein Task fertig ist.

TPL - Task Programming Library

Taskmethode: Wait

```
public static void Main(String[] args){
    // Anlegen und Ausfuehren einer Task
    var task = Task.Run(
        () => {
            Console.WriteLine("alora .. ");
        }
    );

    // Erst durch den Aufruf der Wait Methode
    // wird alora in der Konsole ausgegeben.
    task.Wait();
}
```

TPL - Task Programming Library

Taskmehtode: Result

Die `Result` Methode einer Task blockiert den umgebenden Thread, bis eine Task das geforderte **Ergebnis** berechnet hat.

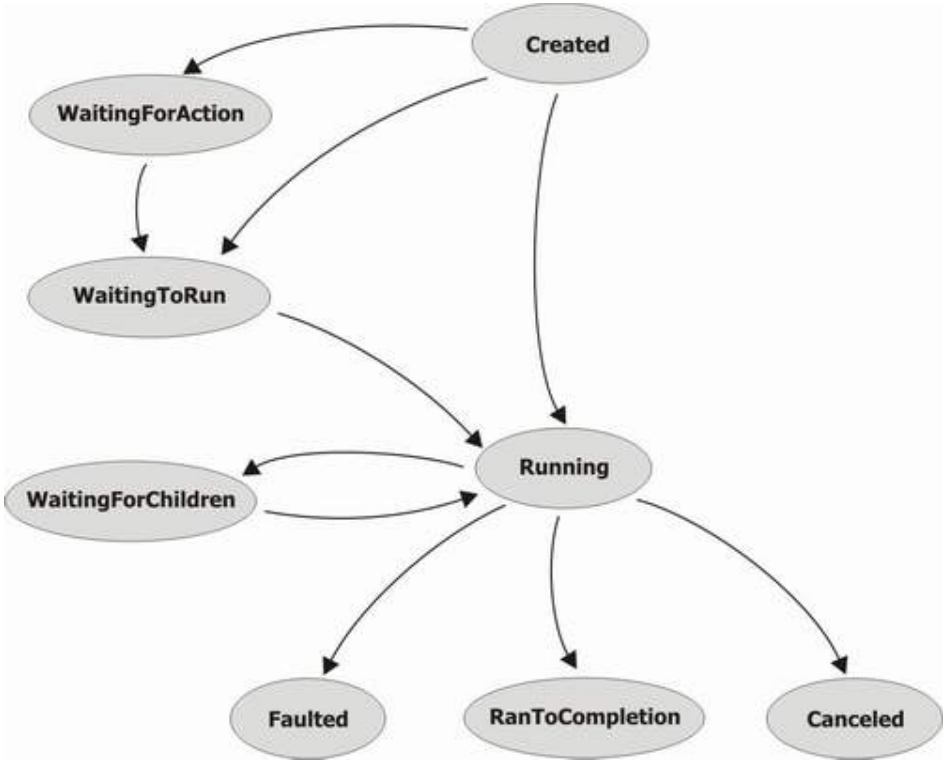
TPL - Task Programming Library

Taskzustände

```
public static void Main(String[] args){
    var task = new Task (
        () => {
            Console.WriteLine("alora .. ");
        }
    );

    Console.WriteLine(task.Status); // Created
    task.Start();
    Console.WriteLine(task.Status); // Ran
    Task.Wait();
    Console.WriteLine(task.Status); // RanToCompletion
}
```

TPL - Task Programming Library

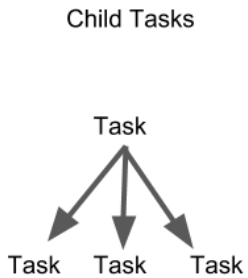
Taskstatemachine

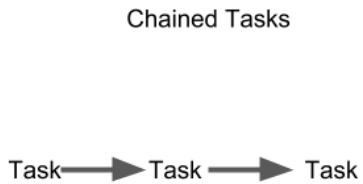


TPL - Task Programming Library

Task Continuation

Task können mit anderen Tasks in Relation gesetzt werden, um als **Einheit** ausgeführt werden zu können.

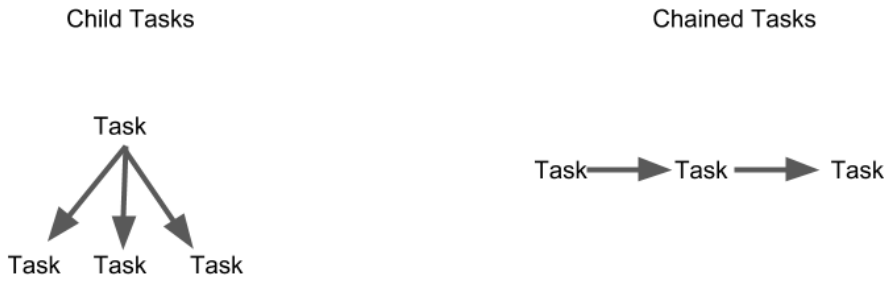




TPL - Task Programming Library

Task Continuation - Chained Tasks

Es ist möglich eine **Kette von Task** zu defienieren, um sie in einer bestimmten Reihenfolge auszuführen.



TPL - Task Programming Library

Task Continuation - Chained Task

```
public static void Chain(){
    Task<int> starter = new Task<int>(
       () => {
          return 42;
       }
    );

    var success = starter.ContinueWith(
        t => {
           Console.WriteLine(t.Result);
           Console.WriteLine(t.Status);
        },
        TaskContinuationOptions.OnlyOnRanToCompletion
    );
   ...
```

TPL - Task Programming Library

Task Continuation - Chained Task

```
    var failure = starter.ContinueWith(
       t => {
           Console.WriteLine(t.Result);
           Console.WriteLine(t.Status);
       },
       TaskContinuationOptions.OnlyOnFaulted
    );

    starter.Start();
    try{
       starter.Wait();
    }catch(SystemException e){
       ...
    }
}
```

TPL - Task Programming Library

Task Continuation - Chained Task

```
var task = Task.Run (
     () => { return 42; }
 );

 var data = task.Result;
 var successor = Task.Run(
    () => {
        Console.WriteLine(data);
        Console.WriteLine(task.Status);
    }
 );

 task.Wait();
```

TPL - Task Programming Library

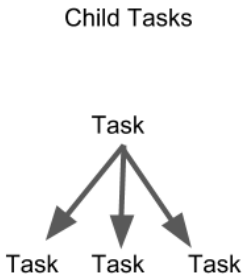
Task Continuation - Chained Task

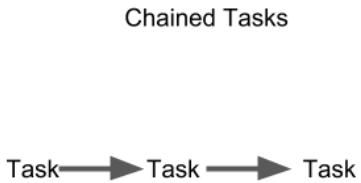
```
var data = await Task.Run (
     () => { return 42; }
 );

 var successor = await Task.Run(
      () => {
         Console.WriteLine(data);
      }
 );
```

TPL - Task Programming Library

Task Continuation - Child Task

Eine Task wird erst ausgeführt, wenn alle **Child Tasks** ihre Ausführung beendet haben.





TPL - Task Programming Library

Task Continuation - Child Task

```
var task = Task.WhenAll(
      Task.Run(() => 10),
      Task.Run(() => 21),
      Task.Run(() => 14)
);

var data = task.Result;
data.ToList().ForEach(Console.WriteLine);
```

TPL - Task Programming Library

Task Continuation - Child Task

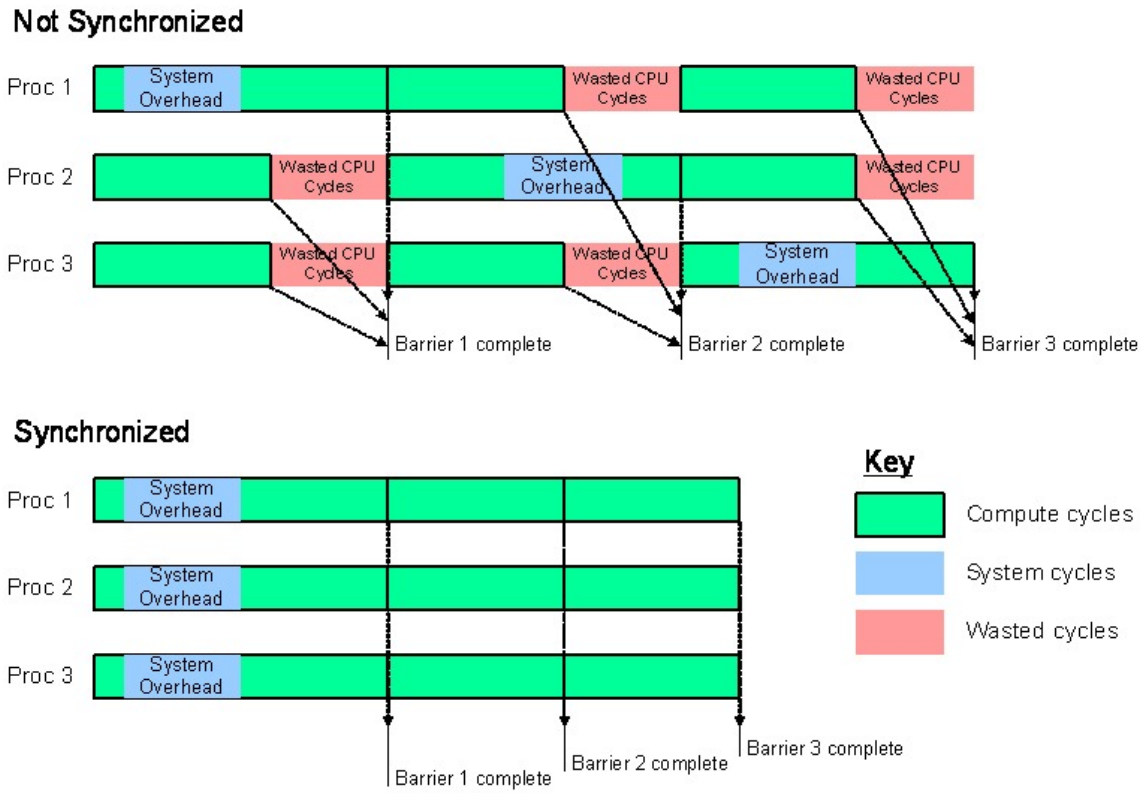
```
var data = await Task.WhenAll(
      Task.Run(() => 10),
      Task.Run(() => 21),
      Task.Run(() => 14)
);

data.ToList().ForEach(Console.WriteLine);
```

Threadsynchronisation - Semaphoren

In der Programmierung versteht man unter Threadsynchronisation die Koordinierung des zeitlichen Ablaufs mehrerer nebenläufiger Threads.

Hinweis: Threads tauschen im Laufe einer Threadsynchronisation keine Daten aus.

Threadsynchronisation - Semaphoren



Threadsynchronisation - Semaphoren

Eine Semaphore ist eine Datenstruktur zur Synchronisierung von Threads.

- Eine Semaphore verwaltet eine Reihe Permits.
- Wird auf den Semaphor zugegriffen wird die Zahl von Permits um eins verringert.
- Stehen keine Permits mehr zur Verfügung, wird der Thread in dem der Zugriff erfolgt blockiert.

Threadsynchronisation - Semaphoren

```
// Anlegen einer Semaphore
SemaphoreSlim sem = new SemaphoreSlim(permitCount: 0);

// Mit dem Aufruf der Wait Methode wird die Zahl der
// Permits um eins verringert. Sind keine Permits mehr
// vorhanden wird der die Methode aufrufende Thread
// blockiert.
sem.Wait();
...
// Durch den Aufruf der Release Methode wird ein Permit
// an die Semaphore zurueckgegeben.
sem.Release();
```

Threadsynchronisation - Semaphoren

```
public class Crane {
    public static readonly SemaphoreSlim CraneGuard = new
        SemaphoreSlim(0);
    public static void Run(){
        while(true){
            Move("Storage", "MachineA");
            MachineA.MachineAGuard.Release();
            CraneGuard.Wait();
            Move("MachineA", "MachineB");
            MachineB.MachineBGuard.Release();
            CraneGuard.Wait();
            Move("MachineB", "Storage");
            Console.WriteLine("...");
        }
    }
}
```

Threadsynchronisation - Semaphoren

```
public class MachineA {
    public static readonly SemaphoreSlim MachineAGuard =
        new SemaphoreSlim(0);

    public static void Run(){
        while(true) {
            MachineAGuard.Wait();
            Process();
            Crane.CraneGuard.Release();
        }
    }
}
```

Threadsynchronisation - Semaphoren

```
public class MachineB {
    public static readonly SemaphoreSlim MachineBGuard =
        new SemaphoreSlim(0);

    public static void Run(){
        while(true) {
            MachineBGuard.Wait();
            Process();
            Crane.CraneGuard.Release();
        }
    }

}
```

Threadsynchronisation - Barrier

```
public class Logger {
    public readonly static System.Threading.Barrier
        Barrier = new System.Threading.Barrier(5);

    public void Run() {
        Barrier.SignalAndWait();
        Console.WriteLine("echo");
    }

}
```

Threadsynchronisation - Barrier

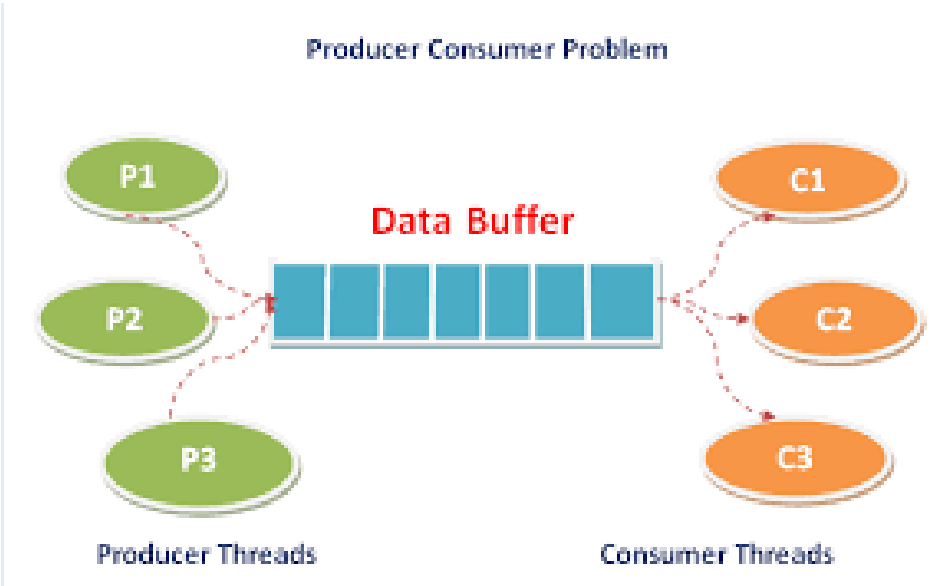
```
await Task.WhenAll(
    Task.Run(new Logger().Run),
    Task.Run(new Logger().Run),
    Task.Run(new Logger().Run),
    Task.Run(new Logger().Run),
    Task.Run(new Logger().Run)
);
```

Threadcommunikation - BlockingCollection

Für die Kommunikation unter Threads muss in einer Anwendung ein Shared Memory zur Verfügung gestellt werden. Damit können mehrere Threads auf einen gemeinsamen Datenspeicher zugreifen.

Für Interthreadkommunikation wird dabei gerne auf das Producer/Consumer Muster zurückgegriffen.

Threadcommunikation - BlockingCollection



Threadcommunikation - BlockingCollection

Die .net Spezfikation realisiert Interthreadkommunikation mit BlockingCollections.

BlockingCollections funktionieren dabei nach dem FIFO Prinzip.

Threadcommunikation - BlockingCollection

```
public class Crane {
    public static BlockingCollection<int> StorageQueue =
        new BlockingCollection<int>(100);

    public void Run() {
        while (true) {
            var item = StorageQueue.Take();

            Console.WriteLine($"storage queue take:
                {item}");
            MachineA.MachineAQueue.Add(item);
        }
    }
}
```

Threadcommunikation - BlockingCollection

```
public class MachineA {
    public readonly static BlockingCollection<int>
        MachineAQueue = new BlockingCollection<int>(100);
    private readonly string _name;

    public void Run() {
        while (true) {
            var item = MachineAQueue.Take();
            Console.WriteLine($"Machine A-{_name}
                processed: {item}");
            Task.Delay(400);
            MachineB.MachineBQueue.Add(item);
        }
    }
}
```

Threadcommunikation - BlockingCollection

```
public class MachineB {
    private readonly string _name;
    public static readonly BlockingCollection<int>
        MachineBQueue = new BlockingCollection<int>(100);

    public void Run() {
        while (true) {
            var item = MachineBQueue.Take();
            Task.Delay(100);
            Console.WriteLine($"Machine B-{_name}
                processed: {item}");
        }
    }

}
```

Threadcommunikation - BlockingCollection

```
Console.WriteLine("Threadcommunication -
    BlockingCollection");

int[] data = { 5, 5, 2, 1, 3 };

data.ToList().ForEach(Crane.StorageQueue.Add);
await Task.WhenAll(
    Task.Run(new Crane().Run),
    Task.Run(new MachineA("IK_1").Run),
    Task.Run(new MachineA("IK_2").Run),
    Task.Run(new MachineB("AL").Run)
);
```