3.A: Was ist ein Programm? Was ist ein Prozess? Was ist der Zusammenhang zwischen Prozess und Programm?

Programm:
Ein Programm ist eine statische Menge von Anweisungen, die von einem Computer ausgeführt werden sollen. Es ist eine Datei oder ein Satz von Dateien, die Anweisungen, Daten und Ressourcen enthalten, die vom Betriebssystem und der CPU verwendet werden, um eine bestimmte Aufgabe oder Funktion zu erfüllen. Ein Programm liegt in inaktiver Form auf der Festplatte vor und wird erst aktiv, wenn es in den Arbeitsspeicher geladen und ausgeführt wird.

- **Beispiel für ein Programm:** Microsoft Word ist ein Programm, das eine Textverarbeitungsfunktion bereitstellt. Die Anweisungen und Daten, die zum Starten und Verwenden von Microsoft Word erforderlich sind, werden als Dateien auf der Festplatte gespeichert.

Prozess:
Ein Prozess ist eine Instanz eines Programms, das derzeit ausgeführt wird. Wenn ein Programm gestartet wird, erstellt das Betriebssystem einen Prozess, der die Ausführung des Programms übernimmt. Jeder Prozess hat seine eigenen Systemressourcen, wie zum Beispiel Speicher, Dateideskriptoren und CPU-Zeit, und läuft in seinem eigenen Adressraum. Ein Prozess besteht also aus einem laufenden Programm und den notwendigen Ressourcen, die für seine Ausführung benötigt werden.

- **Beispiel für einen Prozess:** Wenn Sie Microsoft Word starten, wird ein Prozess erstellt, der dieses Programm ausführt. Wenn Sie das Programm zweimal öffnen, werden zwei separate Prozesse erzeugt, die jedoch beide auf dasselbe Programm zurückgreifen.

Unterschiede zwischen Programm und Prozess:

1. **Existenz:**
    - **Programm:** Ein Programm existiert als statische Datei auf einer Festplatte oder einem anderen Speichermedium.
    - **Prozess:** Ein Prozess ist das Ergebnis der Ausführung eines Programms und existiert im Arbeitsspeicher.
2. **Adressraum:**
    - **Programm:** Programme haben keinen eigenen Adressraum, da sie nicht aktiv sind, bis sie ausgeführt werden.
    - **Prozess:** Prozesse haben ihren eigenen Adressraum und erhalten vom Betriebssystem Ressourcen wie CPU-Zeit und Speicher.
3. **Zustandsinformationen:**
    - **Programm:** Programme sind unbewegliche Entitäten, sie verändern sich nicht während ihrer Lagerung.
    - **Prozess:** Prozesse haben Zustände (wie bereit, blockiert, ausgeführt), die sich dynamisch während ihrer Laufzeit ändern.

Beispiel für den Unterschied:
Ein Texteditor wie Microsoft Word ist ein Programm, das auf Ihrer Festplatte gespeichert ist. Sobald Sie das Programm ausführen, startet das Betriebssystem einen Prozess, der Microsoft Word in den Arbeitsspeicher lädt und die Ausführung des Programms startet. Das Programm selbst ist statisch und bleibt unverändert, während der Prozess dynamisch ist und sich ändert, wenn das Programm ausgeführt wird (z.B. wenn es auf Benutzereingaben wartet oder eine Datei öffnet).

3.B: Was ist der PCB? Welche Bedeutung kommt diesem zu bei der Prozessverwaltung?

PCB (Process Control Block):
Der Process Control Block (PCB) ist eine wichtige Datenstruktur im Betriebssystem, die alle Informationen über einen Prozess enthält. Er stellt sicher, dass das Betriebssystem den Status eines Prozesses verfolgen und die Ausführung eines Prozesses verwalten kann. Der PCB wird für jeden Prozess erstellt, wenn dieser erzeugt wird, und enthält alle relevanten Informationen, die für die Ausführung, die Verwaltung und den Wechsel von Prozessen erforderlich sind.

Bedeutung des PCB in der Prozessverwaltung: Der PCB spielt eine entscheidende Rolle bei der Prozessverwaltung, da er alle Informationen speichert, die benötigt werden, um den Zustand eines Prozesses zu verwalten und ihn zu einem späteren Zeitpunkt wieder aufzunehmen. Wenn ein Prozess zum Beispiel unterbrochen wird (z.B. durch einen Kontextwechsel), speichert der PCB den aktuellen Zustand des Prozesses, sodass das Betriebssystem später die Ausführung des Prozesses genau an dieser Stelle fortsetzen kann.

Informationen im PCB:

1. **Prozess-ID (PID):** Eine eindeutige Kennung, die jeden Prozess identifiziert.
2. **Prozesszustand:** Zeigt an, ob sich der Prozess im Zustand „bereit“, „blockiert“, „laufend“ oder einem anderen Zustand befindet.
3. **Program Counter:** Zeigt auf die nächste Anweisung, die der Prozess ausführen wird.
4. **CPU-Register:** Speichert die Inhalte der CPU-Register, wenn ein Prozess unterbrochen wird (wichtig für den Kontextwechsel).
5. **Speicherverwaltung:** Enthält Informationen über den Speicher des Prozesses, wie Basis- und Grenzregister, die den Speicherbereich des Prozesses definieren.
6. **I/O-Status:** Informationen über die offenen Dateien oder Geräte, die der Prozess verwendet.
7. **Priorität:** Gibt an, wie wichtig ein Prozess ist und ob er Vorrang vor anderen Prozessen hat.

Beispiel zur Bedeutung:
Stellen wir uns vor, dass ein Webbrowser und ein Textverarbeitungsprogramm gleichzeitig laufen. Der PCB für jeden Prozess enthält die spezifischen Informationen über diese beiden Prozesse, einschliesslich ihres Zustands und der aktuellen Ausführungsposition. Wenn der Benutzer zwischen diesen Programmen hin- und herschaltet, speichert das Betriebssystem mithilfe des PCB die aktuellen Zustände der Prozesse und kann nahtlos zwischen ihnen wechseln.

Wo wird der PCB verwaltet?
Der PCB wird vom Betriebssystem im Speicher verwaltet. Er wird typischerweise in einem speziellen Speicherbereich gespeichert, der nur für das Betriebssystem zugänglich ist. Wenn ein Prozess erstellt wird, reserviert das Betriebssystem einen PCB und initialisiert ihn mit den erforderlichen Informationen. Während der Prozesslebensdauer wird der PCB ständig aktualisiert, um den aktuellen Zustand des Prozesses widerzuspiegeln.

3.C: Was sind die Aufgaben des Schedulers?

Scheduler:
Der Scheduler ist ein zentraler Bestandteil eines modernen Betriebssystems. Er ist dafür verantwortlich, die CPU-Zeit zwischen den verschiedenen laufenden Prozessen aufzuteilen. In Systemen, in denen mehrere Prozesse oder Threads gleichzeitig laufen, ist der Scheduler dafür verantwortlich, zu entscheiden, welcher Prozess als nächstes CPU-Zeit erhält, wie lange er ausgeführt wird und wann er pausiert oder beendet wird.

Aufgaben des Schedulers:

1. **CPU-Zuteilung:** Der Scheduler entscheidet, welcher Prozess oder Thread die CPU als nächstes erhält. Dies basiert auf verschiedenen Kriterien wie Priorität, Fairness oder dem Scheduling-Algorithmus, der im Betriebssystem implementiert ist.
2. **Kontextwechsel:** Wenn ein Prozess von der CPU verdrängt wird, ist der Scheduler dafür verantwortlich, den Kontext des aktuellen Prozesses zu speichern (Register, Programmzähler, Speicherstatus) und den Kontext des neuen Prozesses zu laden, der die CPU übernimmt.
3. **Effizienz:** Der Scheduler soll sicherstellen, dass die CPU effizient genutzt wird und die Zeit, in der die CPU untätig ist, minimiert wird.
4. **Multitasking:** Der Scheduler unterstützt das gleichzeitige Ausführen mehrerer Prozesse oder Threads (Multitasking), indem er den CPU-Zeitanteil zwischen ihnen aufteilt.
5. **Prioritätenverwaltung:** Der Scheduler kann Prozessen Prioritäten zuweisen, um sicherzustellen, dass wichtigere Aufgaben Vorrang vor weniger wichtigen Aufgaben haben.

Beispiel für den Scheduler:
In einem Betriebssystem wie Windows oder Linux gibt es oft Hunderte von laufenden Prozessen gleichzeitig, wie z.B. das Betriebssystem selbst, Hintergrunddienste und Benutzeranwendungen. Der Scheduler entscheidet dynamisch, welcher Prozess die CPU zu einem bestimmten Zeitpunkt erhält. Dies wird in Millisekundenbruchteilen entschieden. Wenn beispielsweise ein Video abgespielt wird, während ein Programm heruntergeladen wird, kann der Scheduler dafür sorgen, dass das Video flüssig läuft, während der Download im Hintergrund weiterläuft.

Scheduling-Algorithmen:

1. **First-Come, First-Served (FCFS):**
   Dies ist der einfachste Scheduling-Algorithmus, bei dem der erste Prozess, der die CPU anfordert, diese erhält. Es ist nicht sehr effizient in Systemen, in denen Prozesse unterschiedliche Längen haben, da ein langer Prozess die CPU für lange Zeit blockieren kann.
   - **Beispiel:** Eine Druckwarteschlange, bei der der erste Druckauftrag zuerst bearbeitet wird, unabhängig davon, wie lange es dauert.
2. **Round-Robin (RR):**
   Beim Round-Robin-Algorithmus wird jedem Prozess ein Zeitfenster (Zeitscheibe) zugewiesen, in dem er die CPU nutzen kann. Nach Ablauf dieser Zeit wird der Prozess pausiert, und der nächste Prozess erhält die CPU. Dies sorgt für eine faire Verteilung der CPU-Zeit, ist jedoch möglicherweise ineffizient, wenn die Zeitscheiben schlecht gewählt werden.
   - **Beispiel:** In einer Klassenumfrage bekommt jedes Kind eine bestimmte Zeit, um zu sprechen. Wenn die Zeit abgelaufen ist, darf das nächste Kind sprechen.
3. **Shortest Job Next (SJN):**
   Bei diesem Algorithmus erhält der Prozess mit der kürzesten erwarteten Ausführungszeit die CPU. Dies minimiert die Wartezeit für die Prozesse, kann jedoch schwierig zu implementieren sein, da die genaue Ausführungszeit eines Prozesses im Voraus schwer zu bestimmen ist.
   - **Beispiel:** Wenn drei Aufgaben anstehen, und eine dauert 2 Minuten, eine 5 Minuten und eine 10 Minuten, dann wird die 2-Minuten-Aufgabe zuerst ausgeführt.
4. **Priority Scheduling:**
   Prozesse erhalten eine Priorität, und der Scheduler wählt immer den Prozess mit der höchsten Priorität. Niedrig priorisierte Prozesse können jedoch verhungern (d.h. sie werden nie ausgeführt), wenn ständig Prozesse mit höherer Priorität eintreffen.
   - **Beispiel:** Ein Echtzeitsystem, bei dem die Verarbeitung eines kritischen Sensorereignisses Vorrang vor einer Routineberechnung hat.

3.D: Was ist ein Thread? Was ist der Unterschied zwischen einem Thread und einem Prozess? Was haben sie gemeinsam?

Thread:

Ein **Thread** ist die kleinste Ausführungseinheit innerhalb eines Prozesses. Ein Prozess kann aus mehreren Threads bestehen, die parallel oder quasi-parallel laufen. Jeder Thread führt einen Teil des Programmcodes aus, wobei sich alle Threads eines Prozesses denselben Speicher, die Dateien und die anderen Ressourcen des Prozesses teilen. Ein Thread wird auch als "leichtgewichtiger Prozess" bezeichnet, da er weniger Ressourcen benötigt als ein vollständiger Prozess.

- **Vorteil von Threads:** Threads ermöglichen es, mehrere Aufgaben parallel innerhalb eines Prozesses auszuführen. Da sie sich Ressourcen teilen, sind sie schneller zu erstellen und benötigen weniger Speicher als Prozesse.
- **Beispiel:** In einem Webbrowser könnte ein Thread den HTML-Code rendern, während ein anderer die Benutzerinteraktion überwacht und ein dritter Dateien herunterlädt.

Prozess:

Ein **Prozess** ist eine eigenständige Ausführungseinheit, die vom Betriebssystem verwaltet wird. Jeder Prozess besitzt seinen eigenen Speicherbereich, seine eigene Ressourcen (wie Dateien und Netzwerkverbindungen) und ist von anderen Prozessen getrennt. Prozesse können nicht direkt auf den Speicher anderer Prozesse zugreifen, was die Stabilität und Sicherheit des Systems erhöht. Wenn ein Prozess abstürzt, hat dies in der Regel keine Auswirkungen auf andere Prozesse.

- **Vorteil von Prozessen:** Prozesse bieten mehr Isolation und Sicherheit, da sie keinen direkten Zugriff auf die Daten anderer Prozesse haben. Dies verhindert, dass Fehler in einem Prozess andere Prozesse beeinträchtigen.

Unterschiede zwischen Prozessen und Threads:

1. **Speicherverwaltung:**
    - Ein Prozess besitzt seinen eigenen separaten Speicherraum. Ein Absturz eines Prozesses beeinträchtigt andere Prozesse nicht.
    - Threads teilen sich den gleichen Speicherraum innerhalb eines Prozesses. Dies macht die Kommunikation zwischen Threads effizienter, erhöht aber das Risiko von Synchronisationsproblemen wie Race Conditions.
2. **Ressourcen:**
    - Prozesse haben eigene Ressourcen wie Dateideskriptoren und Speicherzuweisungen.
    - Threads teilen sich die Ressourcen des übergeordneten Prozesses, was die Erstellung und Verwaltung von Threads ressourcenschonender macht als die von Prozessen.

3. **Kommunikation:**
    - Die Kommunikation zwischen Prozessen erfordert spezielle Mechanismen, wie Pipes, Shared Memory oder Message Passing, da Prozesse keine gemeinsamen Speicherbereiche haben.
    - Threads kommunizieren durch direkten Zugriff auf gemeinsame Daten im Speicher, was einfacher, aber auch anfälliger für Fehler ist (z.B. Race Conditions).
4. **Kosten:**
    - Die Erstellung eines neuen Prozesses ist im Vergleich zur Erstellung eines neuen Threads teurer, da das Betriebssystem viele Ressourcen zuweisen muss.
    - Threads sind "leichter", da sie keine eigene Kopie des Speichers oder der Ressourcen benötigen.

Gemeinsamkeiten:

- **Zustände:** Sowohl Threads als auch Prozesse können in verschiedenen Zuständen sein, wie "bereit", "laufend" oder "wartend".
- **CPU-Zeit:** Beide benötigen CPU-Zeit, um ihre Aufgaben auszuführen. Das Betriebssystem steuert, wie und wann die CPU den Prozessen oder Threads zugewiesen wird.
- **Planung:** Das Betriebssystem verwendet einen Scheduler, um sowohl Threads als auch Prozessen CPU-Zeit zuzuweisen, basierend auf Priorität und anderen Scheduling-Kriterien.

Beispiel zur Verdeutlichung:

Stellen Sie sich ein Textverarbeitungsprogramm vor. Der **Prozess** des Programms enthält den gesamten Code, der ausgeführt werden muss, um das Programm zu betreiben. Innerhalb dieses Prozesses gibt es jedoch verschiedene **Threads**, die unterschiedliche Aufgaben übernehmen:

1. Ein Thread überwacht die Benutzerinteraktionen (z.B. Tastendrücke und Mausklicks).
2. Ein anderer Thread überprüft die Rechtschreibung im Hintergrund.
3. Ein weiterer Thread rendert den Text auf dem Bildschirm.

Obwohl diese Threads parallel laufen, teilen sie sich den gleichen Speicher und die Ressourcen des Prozesses.

3.E: Was ist eine Race Condition? Was ist ein "Kritischer Abschnitt"? Was sind die ursächlichen Probleme für Race Conditions?

Race Condition:

Eine **Race Condition** tritt auf, wenn der Ausgang eines Programms davon abhängt, in welcher Reihenfolge Threads oder Prozesse auf gemeinsame Ressourcen zugreifen. Dies kann zu inkorrektem Verhalten führen, wenn die Threads nicht korrekt synchronisiert sind.

- **Problem:** Bei einer Race Condition greifen mehrere Threads gleichzeitig auf eine gemeinsame Ressource (z.B. eine Variable) zu, ohne dass sichergestellt ist, dass der Zugriff geordnet abläuft. Dies kann dazu führen, dass verschiedene Threads gleichzeitig lesen und schreiben, was zu inkonsistenten oder unerwarteten Ergebnissen führt.
- **Beispiel:** Angenommen, Sie haben zwei Threads, die auf eine gemeinsame Variable "Konto" zugreifen. Der eine Thread fügt 100 € hinzu, während der andere 50 € abzieht. Wenn die Zugriffe nicht korrekt synchronisiert sind, könnte es sein, dass beide Threads gleichzeitig auf die Variable zugreifen und der endgültige Kontostand falsch ist.

Kritischer Abschnitt:

Ein **kritischer Abschnitt** ist der Teil des Codes, der auf gemeinsam genutzte Ressourcen zugreift und daher geschützt werden muss, um sicherzustellen, dass immer nur ein Thread oder Prozess gleichzeitig darauf zugreifen kann.

- **Problem:** Ohne einen Mechanismus zur Sicherung des kritischen Abschnitts (wie Locks oder Semaphoren) könnten mehrere Threads gleichzeitig versuchen, den gleichen Speicher zu ändern, was zu Race Conditions führt.

Ursachen von Race Conditions:

1. **Fehlende Synchronisation:**
   Wenn mehrere Threads oder Prozesse auf dieselben Ressourcen zugreifen, ohne dass geeignete Synchronisationsmechanismen wie Locks oder Semaphoren eingesetzt werden, kann es zu Race Conditions kommen.
2. **Falsche Nutzung von Synchronisationsmechanismen:**
   Selbst wenn Synchronisationsmechanismen verwendet werden, können Race Conditions auftreten, wenn diese Mechanismen nicht korrekt implementiert sind.
3. **Parallele Ausführung:**
   In Multithreaded-Umgebungen können Threads fast gleichzeitig auf Ressourcen zugreifen, und die Reihenfolge, in der die Operationen ausgeführt werden, ist nicht immer vorhersehbar.

Beispiel für Race Condition:

Angenommen, zwei Bankangestellte verwenden dasselbe Computersystem, um das Konto eines Kunden zu aktualisieren. Der erste Angestellte fügt 100 € hinzu, der zweite zieht gleichzeitig 50 € ab. Ohne Synchronisation könnte das System die beiden Transaktionen gleichzeitig verarbeiten, was zu einem inkorrekten Endsaldo führt.

3.F: Was unterscheidet Low-Level Primitives von High-Level Primitives?

Low-Level Primitives:

Low-Level-Primitives sind grundlegende Synchronisationsmechanismen, die direkt auf der Hardware- oder Betriebssystemebene implementiert sind. Sie bieten eine atomare Ausführung bestimmter Operationen, um sicherzustellen, dass auf kritische Abschnitte nur von einem Thread oder Prozess gleichzeitig zugegriffen wird.

- **Beispiele für Low-Level-Primitives:**
    - **Test-and-Set (TAS):** Eine Operation, die atomar überprüft und ändert, ob eine Ressource verfügbar ist.
    - **Load-Link/Store-Conditional (LL/SC):** Ein Mechanismus, der verwendet wird, um atomare Lese- und Schreiboperationen durchzuführen.
- **Vorteil:** Low-Level-Primitives bieten eine feine Kontrolle über die Synchronisation und sind sehr schnell, da sie oft direkt von der Hardware unterstützt werden.
- **Nachteil:** Sie sind komplexer zu verwenden und anfälliger für Fehler wie Deadlocks oder Race Conditions, wenn sie nicht korrekt implementiert werden.

High-Level Primitives:

High-Level-Primitives abstrahieren die Komplexität der Low-Level-Primitives und bieten benutzerfreundlichere Mechanismen zur Synchronisation und Kommunikation zwischen Threads und Prozessen.

- **Beispiele für High-Level-Primitives:**
    - **Semaphore:** Ein Synchronisationsmechanismus, der eine Zählung verwendet, um den Zugriff auf gemeinsam genutzte Ressourcen zu steuern.
    - **Monitor:** Ein Mechanismus, der die Ausführung von Code in kritischen Abschnitten eines Programms koordiniert, um Race Conditions zu vermeiden.
    - **Message Passing:** Ein Mechanismus, der es Prozessen ermöglicht, Nachrichten auszutauschen, um miteinander zu kommunizieren.
- **Vorteil:** Sie sind einfacher zu implementieren und bieten eine höhere Abstraktionsebene, was das Risiko von Race Conditions verringert.
- **Nachteil:** Sie sind oft langsamer als Low-Level-Primitives, da sie zusätzliche Abstraktionsschichten hinzufügen.

Zusammenhang:

High-Level-Primitives bauen oft auf Low-Level-Primitives auf. Während Low-Level-Primitives die atomare Kontrolle über bestimmte Operationen bieten, verwenden High-Level-Primitives diese, um komplexere Synchronisationsmechanismen bereitzustellen, die für Entwickler einfacher zu verwenden sind.

Co-Existenz:

Low-Level- und High-Level-Primitives können in einem System koexistieren. Während Low-Level-Primitives in Performance-kritischen Bereichen verwendet werden, werden High-Level-Primitives in der Regel bevorzugt, da sie einfacher und sicherer zu verwenden sind.

Beispiel für die Nutzung von Low-Level und High-Level Primitives:

- **Low-Level-Primitiv:** In einem Betriebssystem könnte der **Test-and-Set-Befehl** verwendet werden, um zu überprüfen, ob ein Thread auf eine gemeinsame Ressource zugreifen darf.
- **High-Level-Primitiv:** Ein Entwickler könnte in einem Java-Programm einen **Monitor** verwenden, um sicherzustellen, dass nur ein Thread gleichzeitig eine Methode ausführt, die auf gemeinsame Ressourcen zugreift.

3.G: Was ist ein Semaphore? Welche Arten von Semaphoren gibt es? Wie funktioniert ein Semaphore?

Semaphore:

Ein **Semaphore** ist ein Synchronisationsmechanismus, der verwendet wird, um den Zugriff auf gemeinsam genutzte Ressourcen zu steuern. Es handelt sich um eine Variable, die verwendet wird, um zu zählen, wie viele Prozesse oder Threads auf eine Ressource zugreifen dürfen.

- **Funktionsweise:** Semaphoren verwenden zwei Operationen: **wait()** (auch als P-Operation bezeichnet) und **signal()** (auch als V-Operation bezeichnet).
    - **wait():** Reduziert den Wert des Semaphores, wenn die Ressource verfügbar ist.
    - **signal():** Erhöht den Wert des Semaphores, wenn eine Ressource freigegeben wird.

Arten von Semaphoren:

1. **Binärer Semaphore:**
   Ein binärer Semaphore ist ein Semaphore, der nur die Werte 0 oder 1 annehmen kann. Er wird verwendet, um den Zugriff auf eine Ressource zu steuern, die nur von einem Thread oder Prozess gleichzeitig verwendet werden darf. Dies ist vergleichbar mit einem **Mutex**.
2. **Zählender Semaphore (Counting Semaphore):**
   Ein zählender Semaphore kann einen beliebigen Wert annehmen und wird verwendet, um den Zugriff auf eine Ressource zu steuern, die von mehreren Threads gleichzeitig verwendet werden darf. Der Wert des Semaphores gibt an, wie viele Threads gleichzeitig auf die Ressource zugreifen dürfen.

Funktionsweise eines Semaphores:

Wenn ein Thread oder Prozess auf eine Ressource zugreifen möchte, ruft er die **wait()**-Operation auf dem Semaphore auf:

- Wenn der Wert des Semaphores grösser als 0 ist, bedeutet dies, dass die Ressource verfügbar ist, und der Semaphore wird um 1 verringert.
- Wenn der Wert 0 ist, muss der Thread oder Prozess warten, bis eine andere Ressource freigegeben wird.

Sobald ein Thread die Ressource nicht mehr benötigt, ruft er die **signal()**-Operation auf, um den Wert des Semaphores zu erhöhen und möglicherweise einen wartenden Thread aufzuwecken.

Beispiel:

Stellen Sie sich einen Drucker in einem Büro vor, der von mehreren Computern genutzt wird. Der Semaphore könnte auf die Anzahl der verfügbaren Druckaufträge gesetzt werden, die der Drucker gleichzeitig verarbeiten kann. Jeder Computer, der einen Druckauftrag senden möchte, würde **wait()** aufrufen, bevor der Druckvorgang startet. Wenn der Drucker beschäftigt ist, müsste der Computer warten. Sobald der Drucker den Auftrag abgeschlossen hat, wird **signal()** aufgerufen, und ein wartender Auftrag kann verarbeitet werden.