Probleme

- Problematisch an der Implementierung des Beispiels auf der letzten Folie ist, dass der Zeiger auf den Auftrag `job` nicht geschützt ist und es so zu fehlerhaften Ausführungen kommen kann.
- Durch Verwendung eines zusätzlichen Semaphors kann dieses Problem behoben werden.
- Ist die Zeit zwischen zwei Aufträgen zu kurz um die rechtzeitige Bearbeitung sicherzustellen, so kann es zu weiteren Problemen kommen:
  - Problem 1: Der Prozess **Contractor** muss warten, weil der Prozeß **Worker** den letzten Auftrag noch bearbeitet.
  - Problem 2: Der letzte Auftrag wird überschrieben, falls dieser noch gar nicht bearbeitet wurde. Abhängig von der Implementierung des Semaphors könnte dann der neue Auftrag zudem zweifach ausgeführt werden.

→ mit Semaphoren sind nur einfache Signalisierungsprobleme (ohne Datentransfer) zu lösen, ansonsten sollten Warteschlangen verwendet werden

Signalisierung durch Semaphore: Leser-Schreiber-Beispiel

- Vorherige Lösung:

Reader:

```
...
        down(semWriter);
        down(semCounter);
        rcounter++;
        up(semCounter);
        up(semWriter);

        read();

        down(semCounter);
        rcounter--;
        up(semCounter);
...
```

Writer:

```
...
        down(semWriter);

        while(true)
        {
                down(semCounter);
                if(rcounter==0)
                        break;
                up(semCounter);
        }
        up(semCounter);

        write();

        up(semWriter);
...
```

Problem: Busy Waiting

Signalisierung durch Semaphore: Leser-Schreiber-Beispiel

- Lösung mit Signalisierung:

Reader:

```
...
            down(semWriter);
            down(semCounter);
            rcounter++;
            if(rcounter==1)
                        down(semReader);
            up(semCounter);
            up(semWriter);

            read();

            down(semCounter);
            rcounter--;
            if(rcounter==0)
                        up(semReader);
            up(semCounter);
...
```

Writer:

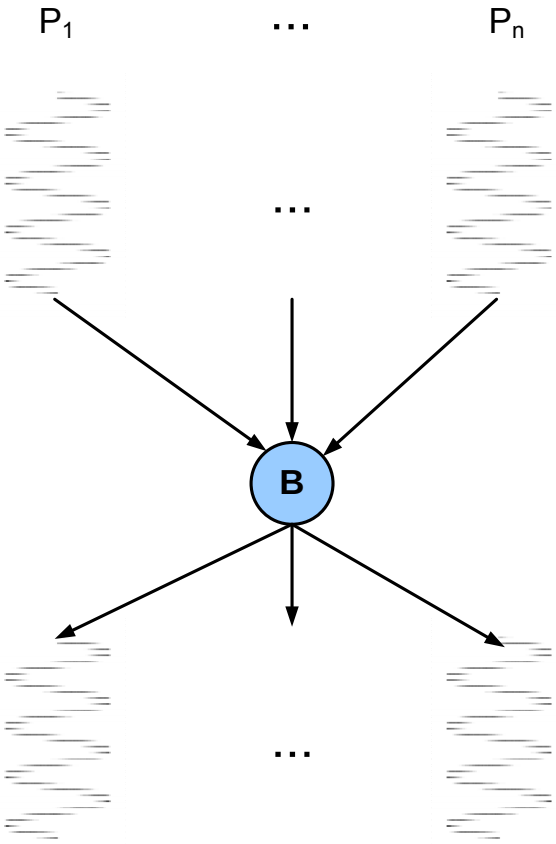
```
...
            down(semWriter);
            down(semReader);
            up(semReader);

            write();


            up(semWriter);
...
```

Nebenläufigkeit

Synchrone Kommunikation: Barrieren, Occam

Synchrone Kommunikation: Barrieren

- **Definition**: Eine Barriere für eine Menge *M* von Prozessen ist ein Punkt, den alle Prozesse $P_i \in M$ erreichen müssen, bevor irgendein Prozess aus *M* die Berechnung über diesen Punkt hinaus fortfahren kann.
- Der Spezialfall für $|M|=2$ wird als Rendezvous, siehe auch Ada, bezeichnet.
- Barrieren können mit Hilfe von Semaphoren implementiert werden.



Occam

- Als Programmiersprache wurde Occam verwendet, mit der man parallel Abläufe festlegen konnte.
- Als Namenspate fungierte der Philosoph William of Ockham. Sein Postulat „*Dinge sollten nicht komplizierter als unbedingt notwendig gemacht werden*“ war Motto der Entwicklung.
- Occam basiert auf dem Modell CSP (communicating sequential processes) von C.A.R. Hoare; siehe auch CCS (Calculus of Communicating Systems) von R. Milner
- Occam ist eine Sprache, die die parallele Ausführung von Aktionen direkt mit einbezieht
- Die Kommunikation zwischen den einzelnen Prozessen erfolgt synchron über unidirektionale Kanäle.
- Die Realisierung auf dem Transputer ist 1:1. Als Kanal zwischen zwei Prozessen auf unterschiedlichen Transputern kann ein (halber) Link benutzt werden. Befinden sich die beiden Prozesse auf einem Transputer, so kann der Kanal über Speicherplätze simuliert werden.
- Siehe http://vl.fmnet.info/occam/

William of Ockham

C.A.R. Hoare

Occam

- Code wird in Occam zu Blöcken zusammengefasst, indem die einzelnen Zeilen alle gleichweit eingerückt werden
- Eine Anweisung wird durch das Ende der Zeile beendet
- Sprachelemente:
  - Eingabe ? :
  - Ausgabe !:
  - Sequentielle Ausführung SEQ:
  - Parallele Ausführung PAR:
  - Alternative Ausführung ALT*:

```
keyboard ? c
```

```
screen ! c
```

```
SEQ
    x:=1
    y:=2
```

```
PAR
    keyboard ? x
    screen ! y
```

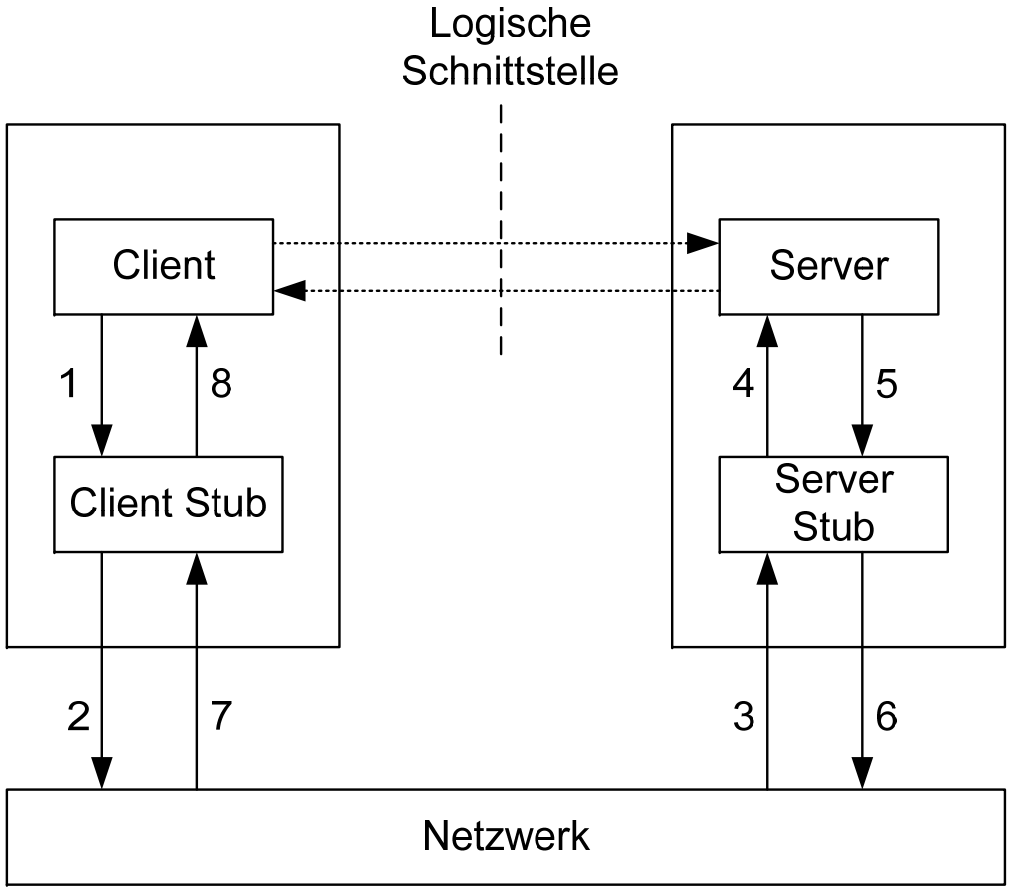
```
ALT
    x<10 & chan1 ? y
        screen ! y
    x<20 & chan2 ? y
        screen ! y
```

*Bei der ALT kann für jeden Block eine Bedingung, sowie eine Eingabe (beide optional) angegeben werden. Es wird derjenige Block ausgeführt, dessen Bedingung wahr ist und auf dem Daten eingehen. Trifft dies für mehrere Blöcke zu, so wird ein Block gewählt und ausgeführt.

Nebenläufigkeit

Funktionsaufrufe als Kommunikation

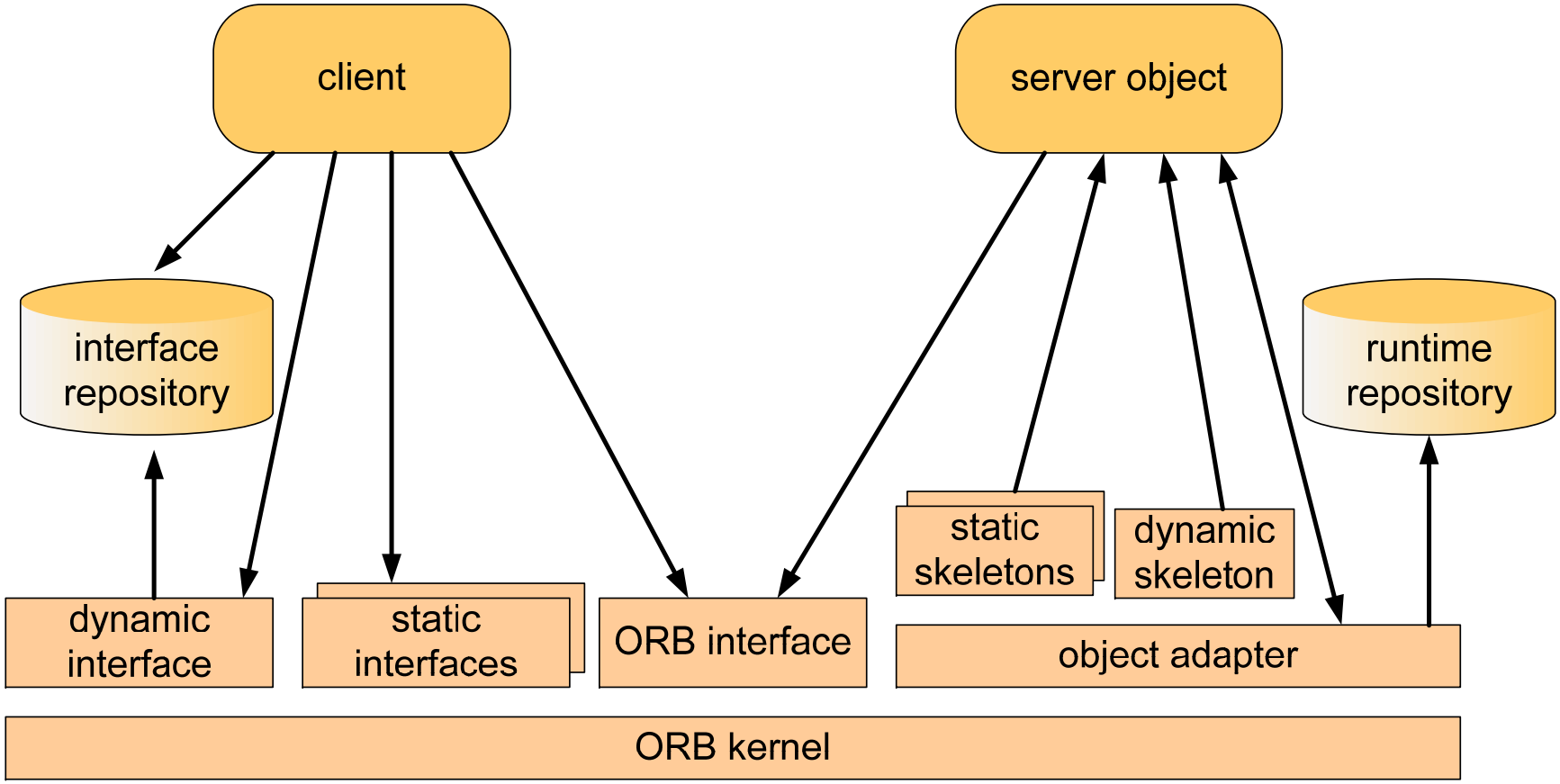
Remote Procedure Call (RPC)



Ablauf RPC

- Bei einem Funktionsaufruf über RPC werden folgende Schritte ausgeführt:
  1. Lokaler Funktionsaufruf vom Client an Client Stub
  2. Konvertierung des Funktionsaufrufs in Übertragungsformat und Senden der Nachricht
  3. Empfang der Nachricht von Kommunikationschicht
  4. Entpacken der Nachricht und lokaler Funktionsaufruf
  5. Übermittlung des Ergebnisses von Server an Server Stub
  6. Konvertierung des Funktionsergebnisses in Übertragungsformat und Senden der Nachricht
  7. Empfang der Nachricht von Kommunikationschicht
  8. Entpacken der Nachricht und Übermittlung des Ergebnisses an Client
- Voraussetzung für Echtzeitfähigkeit: Echtzeitfähiges Kommunikationsprotokoll und Mechanismus zum Umgang mit Nachrichtenverlust

Corba (Common Object Request Broker Architecture)



Komponenten in Corba

- **ORB (Object Request Broker)**: vermittelt Anfragen zwischen Server und Client, managt die Übertragung, mittlerweile sind auch echtzeitfähige ORBs verfügbar
- **ORB Interface**: Schnittstelle für Systemdienstaufrufe
- **Interface repository**: speichert die Signaturen der zur Verfügung stehenden Schnittstellen, die Schnittstellen werden dabei in der IDL-Notation (Interface Definition Language) gespeichert.
- **Object Adapter**: Überbrückt die Lücke zwischen Corba-Objekten mit IDL-Schnittstelle und Serverobjekten in der jeweiligen Programmiersprache
- **Runtime repository**: enthält die verfügbaren Dienste und die bereit instantiierten Objekte mitsamt den entsprechenden IDs
- **Skeletons**: enthalten die Stubs für die Serverobjektaufrufe

Nebenläufigkeit

Zusammenfassung & Wiederholung

Zusammenfassung

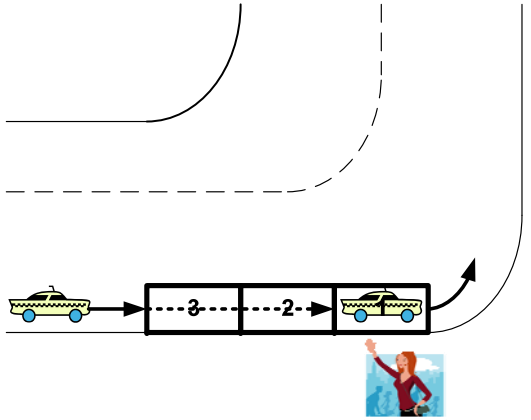
- Folgende Fragen wurden in dieser Vorlesung erklärt und sollten nun verstanden sein:
  - Was ist Nebenläufigkeit / Parallelität?
  - Mit welchen Techniken kann man Nebenläufigkeit erreichen und wann wird welche Technik angewendet?
  - Wie können **race conditions** vermieden werden?
  - Welche Arten der Interprozesskommunikation gibt es (+allgemeine Erklärung)?

Klausurfragen - Nebenläufigkeit

- Klausur WS 06/07
  - Nennen Sie zwei Gründe, wieso nebenläufige Programmierung häufig in Echtzeitsystemen angewandt wird.
  - Was sind Race Conditions?
- Wiederholungsklausur WS 06/07:
  - In der Vorlesung haben Sie die Konzepte Nachrichtenwarteschlangen, Semaphor, sowie Barrieren kennengelernt.
    a) Beschreiben Sie, wie man mit Hilfe von Semaphoren Nachrichtenwarteschlangen implementieren kann.
    b) Beschreiben Sie, wie man mit Hilfe von Nachrichtenwarteschlangen Semaphoren implementieren kann.
    c) Beschreiben Sie, wie man mit Hilfe von Semaphoren Barrieren implementieren kann.
- Klausur WS 07/08
  - **Annahme**: Für folgende Aufgaben können Sie davon ausgehen, dass maximal ein Signal pro Runde an den Automaten geschickt wird. Die Prozesse, die die modellierten Komponenten benutzen, müssen Sie nicht modellieren. Vermeiden Sie Busy Waiting.
  - Modellieren Sie einen Automaten, der das Rendezvouskonzept umsetzt.
  - Modellieren Sie eine Komponente, die das Leser-Schreiber-Problem für 1 Schreiber und beliebig viele Leser mit Schreiberpriorität umsetzt.

Klausur WS07/08 - Nebenläufigkeit

- Gegeben Sie folgendes Szenario: am Münchner Odeonsplatz gibt es eine Wartebucht für Taxis. Zur Vereinfachung gehen wir davon aus, dass die Wartebucht aus drei Plätzen besteht und immer nur ein Passagier gleichzeitig auf ein Taxi wartet. Passagiere steigen an der ersten Wartebucht ein, die Taxis rücken nach, sobald das Taxi vor ihnen losgefahren ist. Implementieren Sie nun schrittweise eine Prozesssynchronisation, so dass es zu keinen Auffahrunfällen kommt, die Taxis in der Ankunftsreihenfolge auch wieder losfahren, Taxis nur mit Passagier losfahren, Passagiere nicht aus Versehen ein nicht-existentes Taxi betreten und es zu keinen Verklemmungen kommt.



a) Notieren Sie die wichtigen Programmabschnitte des Taxiprozesses und des Passagierprozesses. Lassen Sie genügend Platz für spätere Synchronisationsoperationen.
Beispiel: `fahreInErsteWartebucht();`

b) Geben Sie die zur Synchronisation der Taxis und Passagiere benötigten Semaphore, sowie der Initialwerte an. Gehen Sie dabei davon aus, dass zu Beginn kein Taxi in der Wartebucht und keine wartenden Passagiere vorhanden sind.
Beispiel: `semTaxi(1)` würde bedeuten, Sie verwenden einen Semaphor semTaxi, der mit 1 initialisiert ist.
`int i=0;` wenn sie eine ganzzahlige Variable mit Initialisierungswert 1 benutzen wollen.

c) Ergänzen Sie den Taxiprozess und Passagierprozess mit passenden up() und down()-Methoden, um die Aufgabenstellung zu erfüllen.
Beispiel: `down(semTaxi);` bedeutet das Anfordern des Semaphors semTaxi
Beispiel: `up(semTaxi);` bedeutet das Freigeben des Semaphors semTaxi

d) Der Wartebereich am Odeonsplatz ist begrenzt. Stellen Sie sicher, dass maximal 3 Taxis auf Fahrgäste warten und kein Rückstau entsteht. Die Überprüfung ob der Wartebereich belegt ist, soll dabei so schnell wie möglich erfolgen um den Straßenverkehr nicht zu behindern. Andererseits, sollen die Taxifahrer auf jeden Fall in den letzten Warteplatz fahren, falls dieser frei ist.

Kapitel 4

Scheduling