Kurs OMSI
im WiSe 2012/13

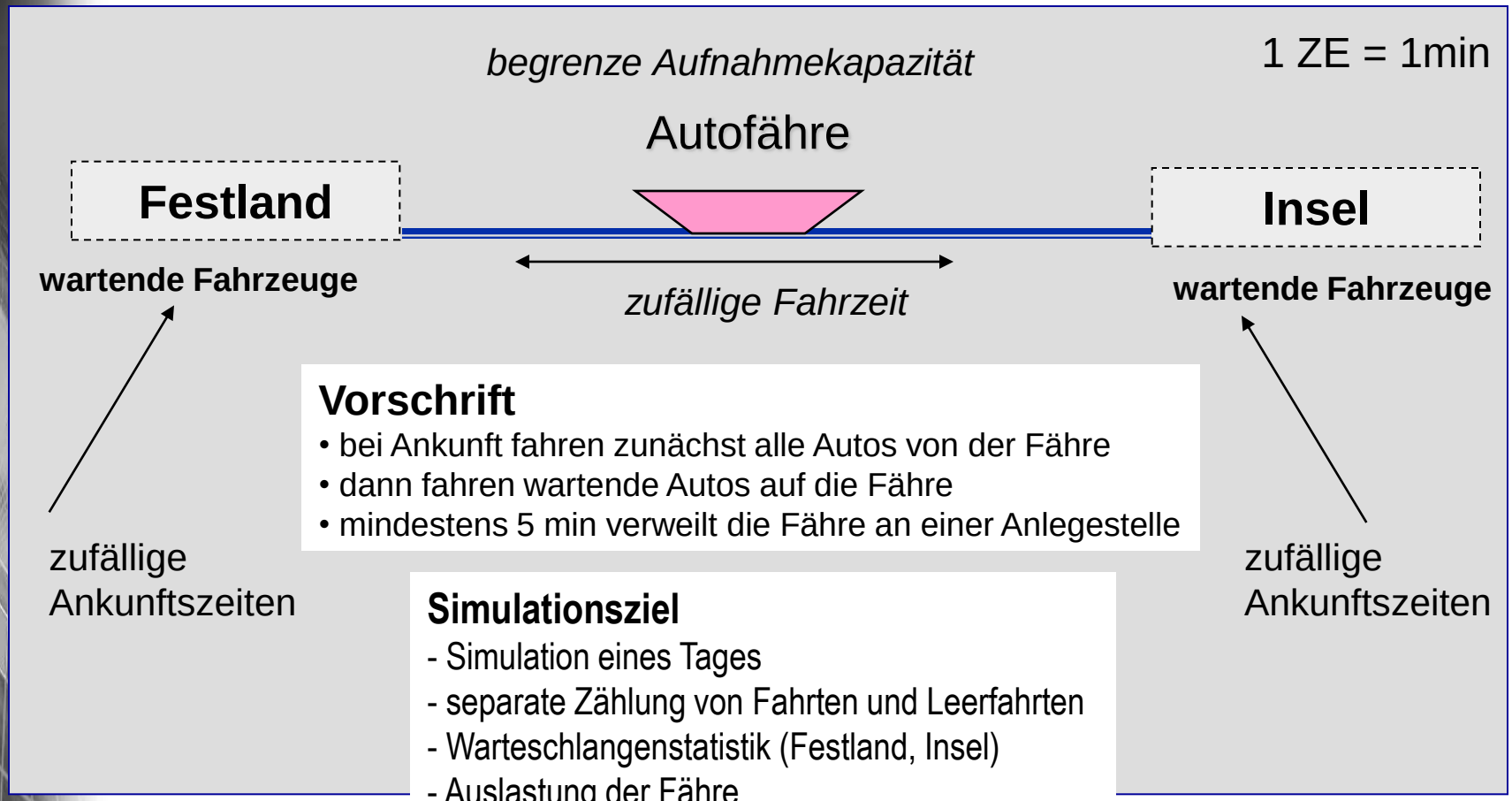
Objektorientierte Simulation mit ODEMx

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

fischer|ahrens|eveslage@informatik.hu-berlin.de

Beispiel: Autofähre

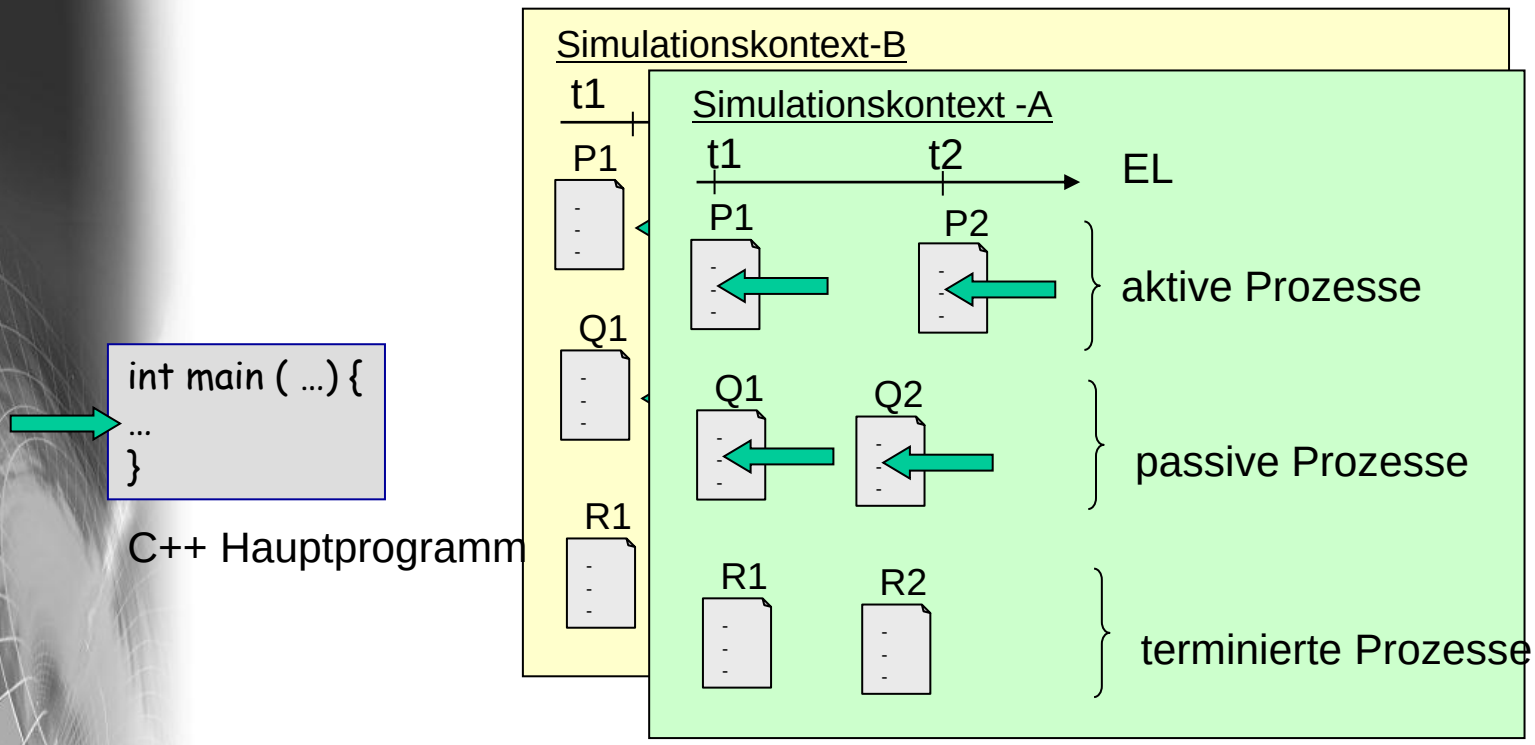
- Wortmodell und informale Darstellung



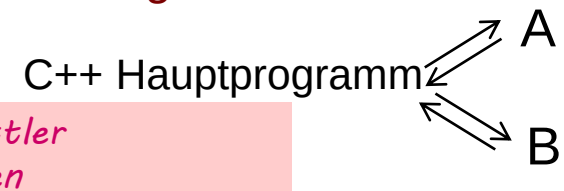
3. Prozessverwaltung

1. Aufgaben von Klasse Simulation
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen (ProcessQueue, Queue)
6. Spezielles Process-Scheduling (Memory)

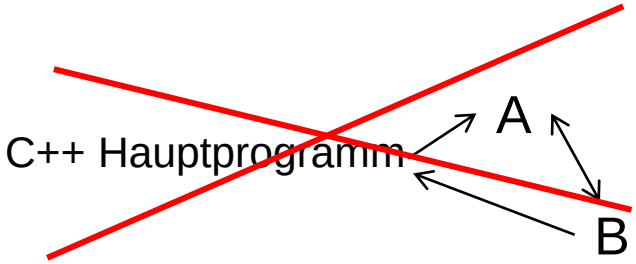
Wdh.: Verwaltung mehrerer Simulationskontexte



Steuerungszenarien:



als Mittler zwischen den Prozess-Systemen



Aufgaben des ODEMx-Hauptprogramms

1. **Systeminitialisierung**
   - Festlegung der Simulationskontexte
   - Bereitstellung aller initialen (zur Modellzeit 0.0) notwendigen Systemelemente des jeweiligen Simulationskontextes
   - Initialisierung der Terminkalender der Simulationskontexte durch Erzeugung und Aktivierung von Process-Objekten

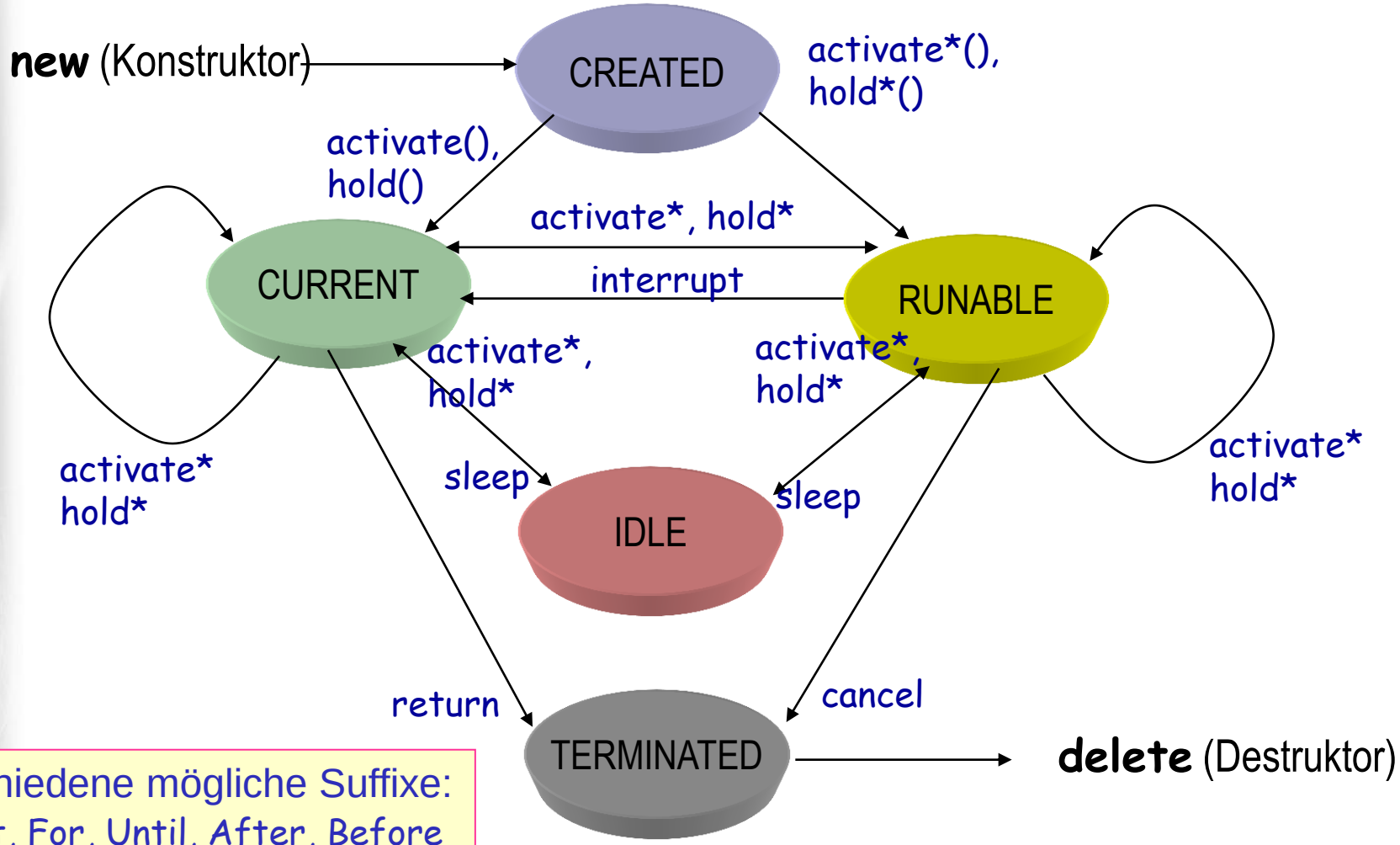
2. **Systemsimulation**
   - Simulationsstart der Simulationskontexte bei Festlegung von Abbruchkonstellation für die Simulation
   - Eventuelle Zwischenreports und Statistische Berechnungen
   - …

3. **Ende der Systemsimulation**

   - Finale Reports und Statische Berechnungen

Hauptprogramm wird nicht wie ein ausgezeichneter Prozess in einem der Terminkalender geführt
(Unterschied: Simula, SLX, früheren Versionen von ODEMx (die nur von einem Simulationskontext ausgingen)

Wdh.: Process-Zustände und Scheduling-Operationen



* verschiedene mögliche Suffixe:
In, At, For, Until, After, Before

Klasse Process: *Member-Funktionen (1)*

```
class Process
        :  public Sched
        ,  public coroutine::Coroutine
        ,  public data::Observable< ProcessObserver > {
public:
    enum ProcessState {
        CREATED, CURRENT, RUNNABLE, IDLE, TERMINATED};

        Process( Simulation& sim, const data::Label& label, ProcessObserver* obs = 0 );
        ~Process();
        ProcessState getProcessState() const;

        void activate();
        void activateIn( SimTime t );
        void activateAt( SimTime t );

        void activateBefore( Sched* s );
        void activateAfter( Sched* s );

        void hold();
        void holdFor( SimTime t );
        void holdUntil( SimTime t );
```

Aktivierung/Reaktivierung:
bei Prioritätsgleichheit als erstmöglicher Eintrag

Aktivierung/Reaktivierung:
mit evtl. Prioritätsanpassung

Aktivierung/Reaktivierung:
bei Prioritätsgleichheit als letztmöglicher Eintrag

Klasse Process: *Member-Funktionen (2)*

```
...
public:
        void sleep();
        virtual void interrupt();
        void cancel();

        Priority getPriority() const;
        Priority setPriority( Priority newPriority );

        SimTime getExecutionTime() const;
        bool isInterrupted() const;
        Sched* getInterrupter();

protected:
        virtual int main() = 0;

public:
        bool hasReturned() const {return validReturn;};
        int getReturnValue() const;
...
```

Blockierung, Unterbrechung, Beendigung

Lesen/Setzen von Prioritäten

Infos zur Ereigniszeit und Unterbrechung

Lebenslauf

Infos zum Lebenslaufende

Process: Scheduling-Operationen (1)

Prozessaktivierungen nach dem LIFO-Prinzip

Achtung: semantischer Unterschied, ob Aufruf vom Hauptprogramm oder einem Prozess-Objekt erfolgt

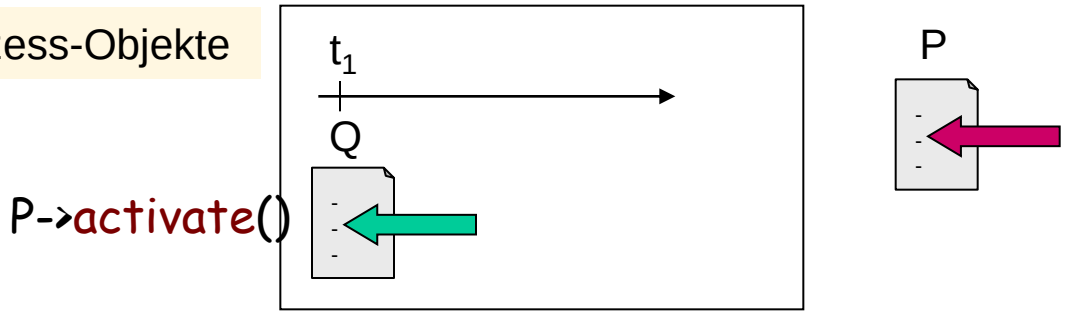
```
void activate();      // Eintrag in ExL zur aktuellen Ereigniszeit now
                      // nach dem LIFO-Prinzip
                      // evtl. Prozesswechsel

void activateIn (SimTime t);
                      // Eintrag in ExL zur Ereigniszeit now + t
                      // nach dem LIFO-Prinzip bei Gleichzeitigkeit und
                      // Prioritätsgleichheit
                      // falls t<0.0, dann t= 0.0

void activateAt (SimTime t);
                      // Eintrag in ExL zur absoluten Ereigniszeit t
                      // nach dem LIFO-Prinzip bei Gleichzeitigkeit und
                      // Prioritätsgleichheit
                      // falls t<now, dann t= now
```

semantisch äquivalent:

P->activate() == P->activateIn(0.0) == P->activateAt(now)

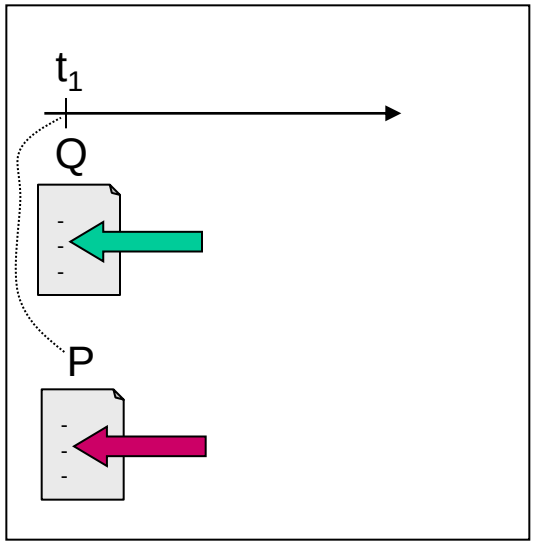
Activate innerhalb eines Simulationskontextes

betrachten 2 Prozess-Objekte



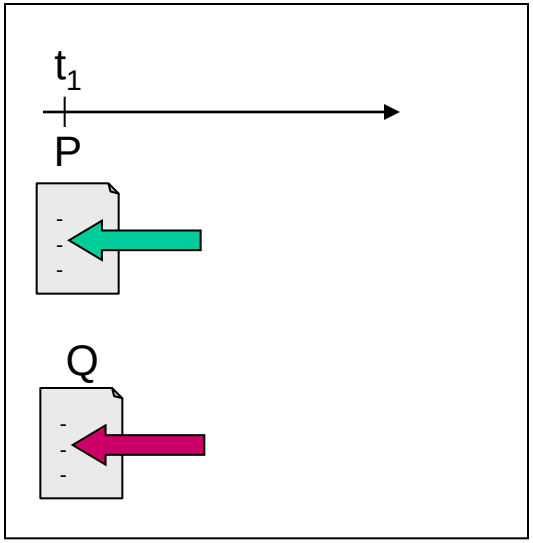
prior(P) < prior(Q)

P hinter Q
→ kein Prozesswechsel



prior(P) >= prior(Q)

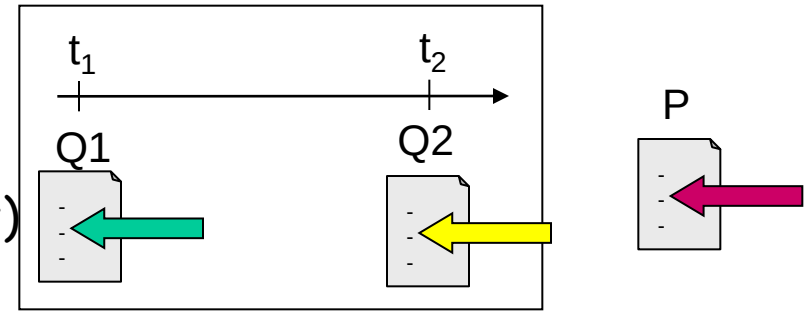
P vor Q
→ mit Prozesswechsel



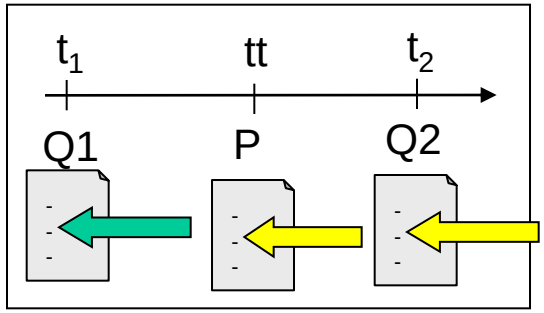
Activate innerhalb eines Simulationskontextes

betrachten 3 Prozess-Objekte

P->activateAt(tt)



Ann.: $t_1 < tt < t_2$



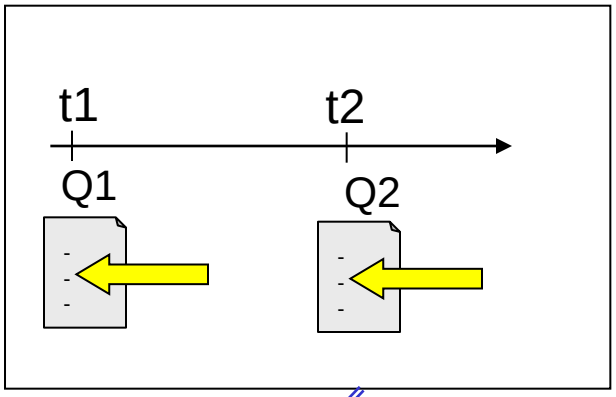
Q1 bleibt vor P
→ kein Prozesswechsel

Activate außerhalb eines Simulationskontextes

betrachten 3 Prozess-Objekte

Simulationskontext (DefaultSimulation-Objekt)

```
int main ( ...) {
  P->activate()
  ...
}
```

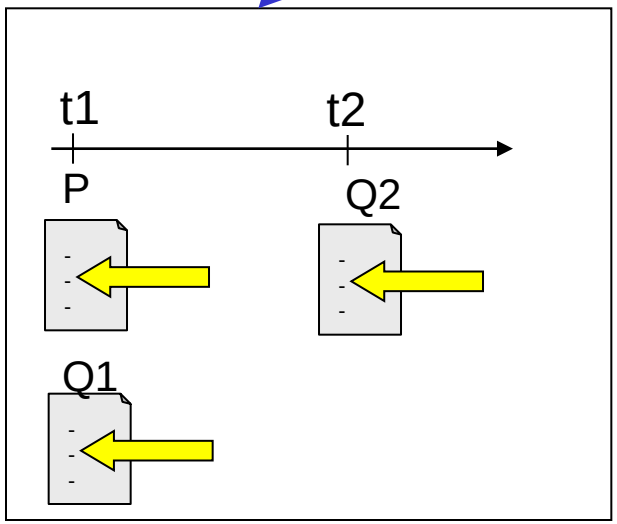


P

als Objekt generieriert

Ann.: prior(P) >= prior(Q1)

```
int main ( ...) {
  ... P->activate()
  ..
}
```



kein Prozesswechsel ! Hauptprogramm setzt Ausführung fort

Process: Scheduling-Operationen (2)

Prozessaktivierungen nach dem Vorher/- Nachherprinzip

```
void activateBefore (Process* p);
                    // unmittelbarer Eintrag vor p mit Ereigniszeit von p,
                    // ggf. Übernahme der Priorität von p
                    // falls p == this: leere Anweisung
                    // falls p nicht in der ExL: Fehlermeldung

void activateAfter (Process* p);
                    // unmittelbarer Eintrag nach p mit Ereigniszeit von p
                    // ggf. Übernahme der Priorität von p
                    // falls p == this: leere Anweisung
                    // falls p nicht in der ExL: Fehlermeldung
```

Process: Scheduling-Operationen (3)

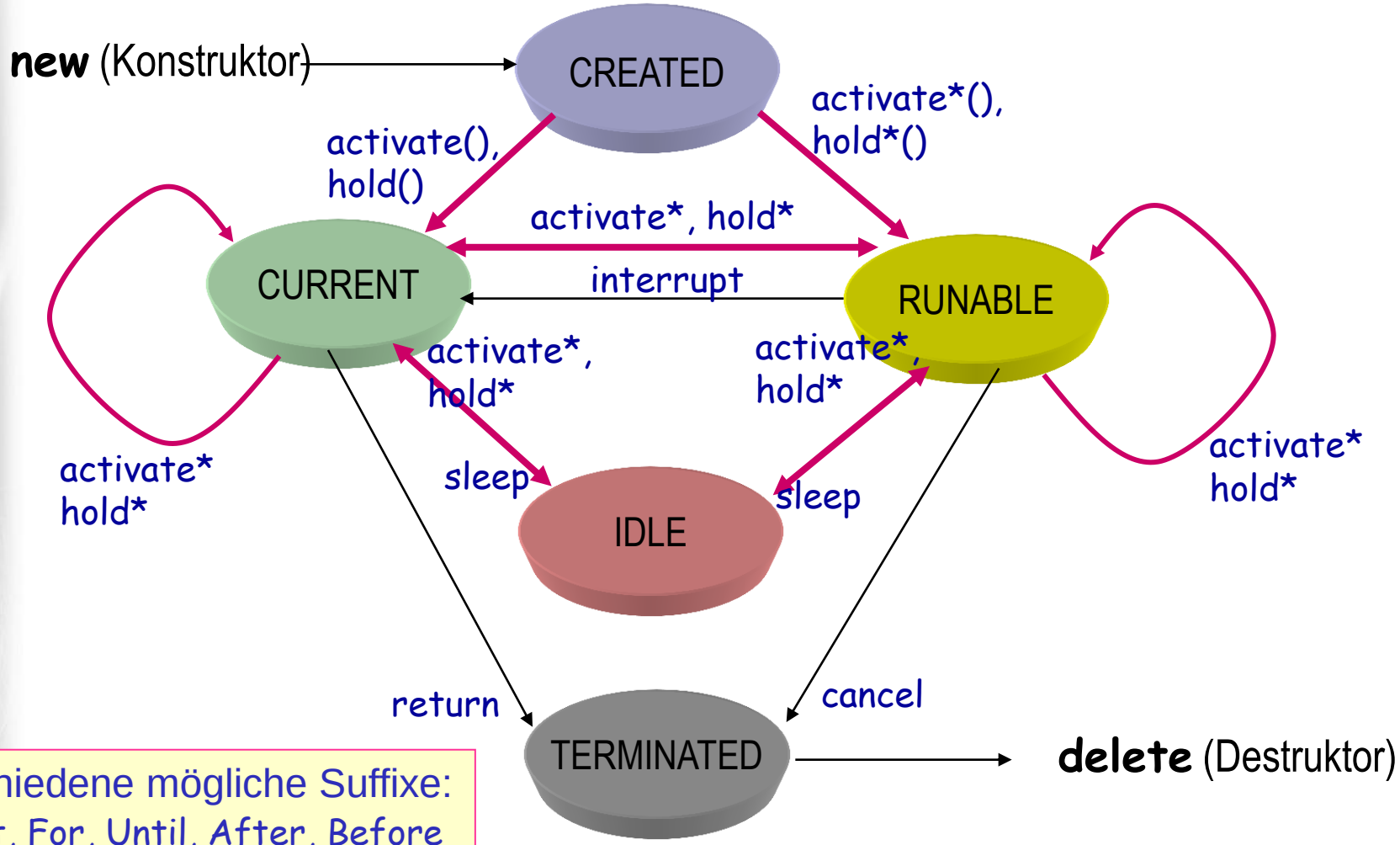
Prozessverzögerungen nach dem FIFO-Prinzip

```
void hold();
                    // Eintrag zur aktuellen Ereigniszeit
                    // als letzter bei gleicher oder niedrigerer Priorität (FIFO)

void holdFor (SimTime t);
                    // Eintrag zur Ereigniszeit now + t
                    // als letzter bei gleicher oder niedrigerer Priorität
                    // falls t<0.0, dann t= 0.0

void holdUntil (SimTime t);
                    // Eintrag zur absoluten Ereigniszeit t
                    // als letzter bei gleicher oder niedrigerer Priorität
                    // falls t<now, dann t= now
```

semantisch äquivalent:

```
p->hold() == p->holdFor(0.0) == p->holdUntil(now)
p->holdUntil(t) == p->holdFor(t-now)
```

Wdh.: Process-Zustände und Scheduling-Operationen



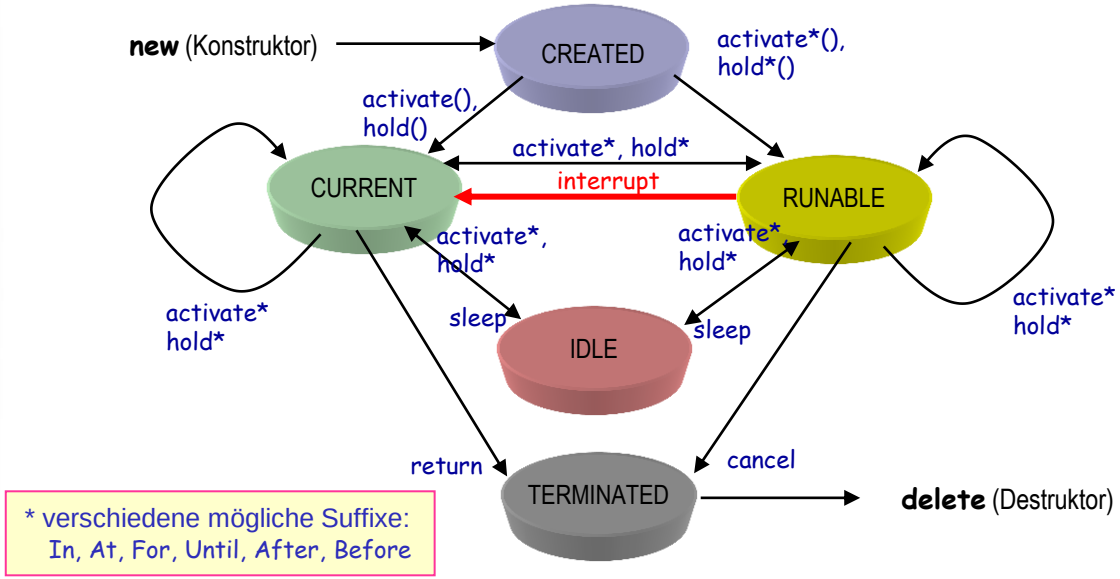
* verschiedene mögliche Suffixe:
In, At, For, Until, After, Before

Process: Scheduling-Operationen (4)

Prozessunterbrechungen

```
void sleep();
        // Entfernung von currentProcess()  / runable-Prozess aus der ExL
        // Zustandswechsel in idle, Ereigniszeit 0.0
        // Aktivierung des ersten ExL-Eintrages
        //          falls Process, dann auch Prozesswechsel )
        //          falls ExL leer, dann Rückkehr ins Hauptprogramm

void cancel();
        // Prozessabbruch, Entfernung aus der ExL
        // Zustandswechsel in terminated
        // erneute Aktivierung führt zum Fehler

virtual void interrupt();
        // Prozessunterbrechung
        // eines Runable-Prozesses
```

Interrupt (detaillierter)

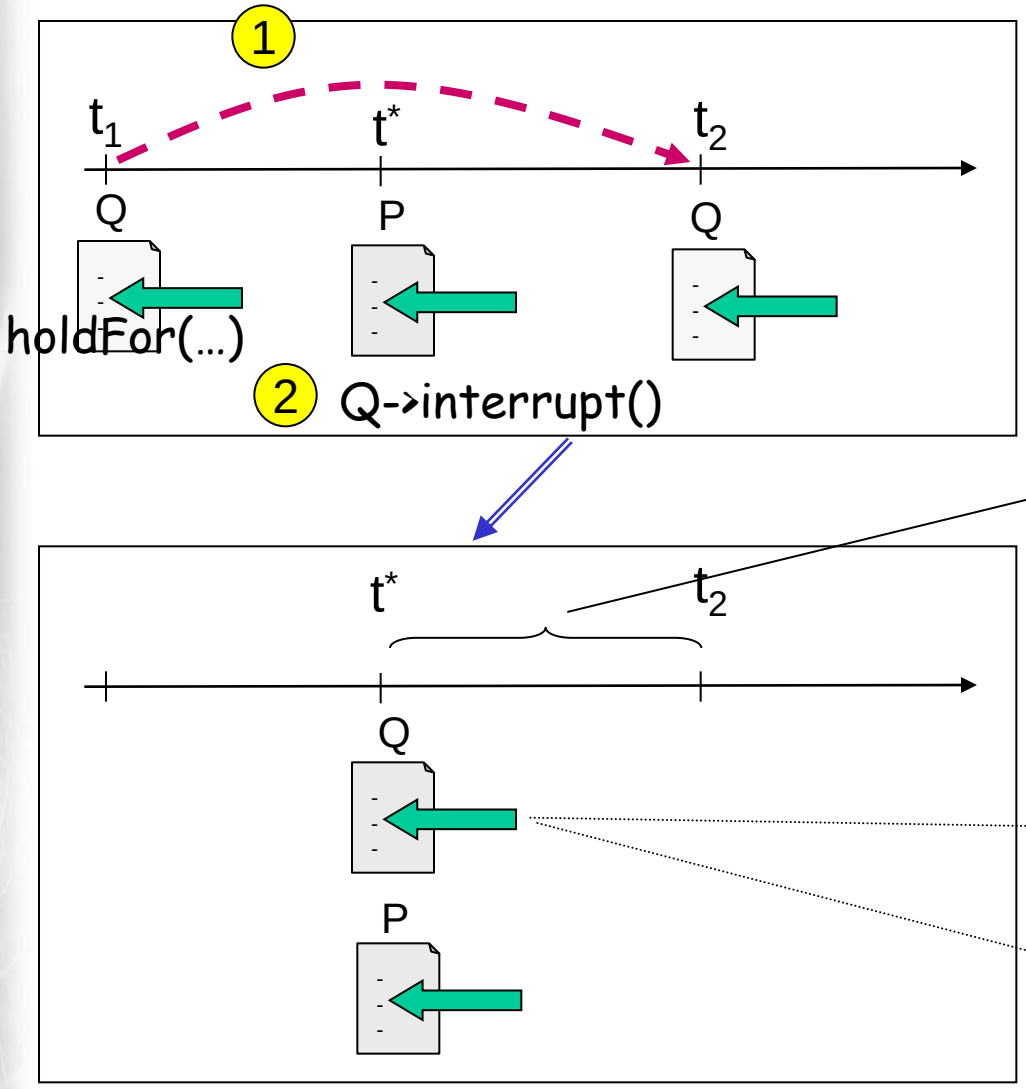


* verschiedene mögliche Suffixe:
In, At, For, Until, After, Before

virtual void interrupt();

- ein runable-Prozess wird vom current-Prozess in seiner hold/activate-Phase unterbrochen
- er wird (aus der Zukunft zurück) zum neuen current-Prozess, trotz eines evtl. bestehenden Prioritätskonfliktes
- bei Fortsetzung seiner Aktivität kann er mit getInterrupter() die erfolgte Unterbrechung erkennen und
- selbst behandeln (z.B. korrigierte Zustandsänderung)

Interrupt (detaillierter)



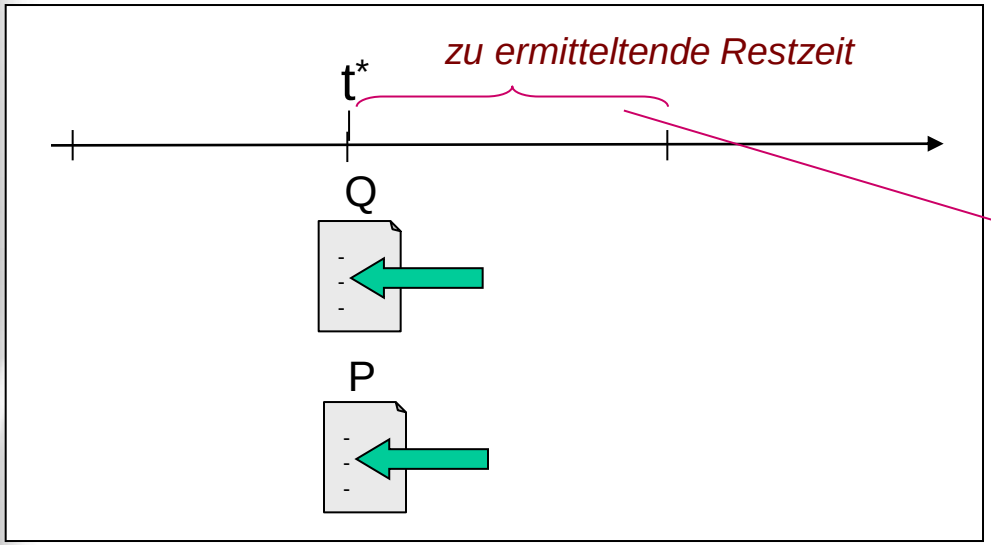
zeitabhängige Zustandsänderung von Q im Intervall $[t_1, t_2]$

konkrete Zustandsänderung ist anwendungsabhängig (Ableitung von Process)

Zeitkorrektur aber keine noch keine Korrektur der Zustandsänderung

```
getInterrupter() == P
// mit Zugriff auf rest

isInterrupted() == true
```

Interrupt (detaillierter)



für Korrektur
der Zustandsänderung
durch Q
wird die „Restzeit“ benötigt

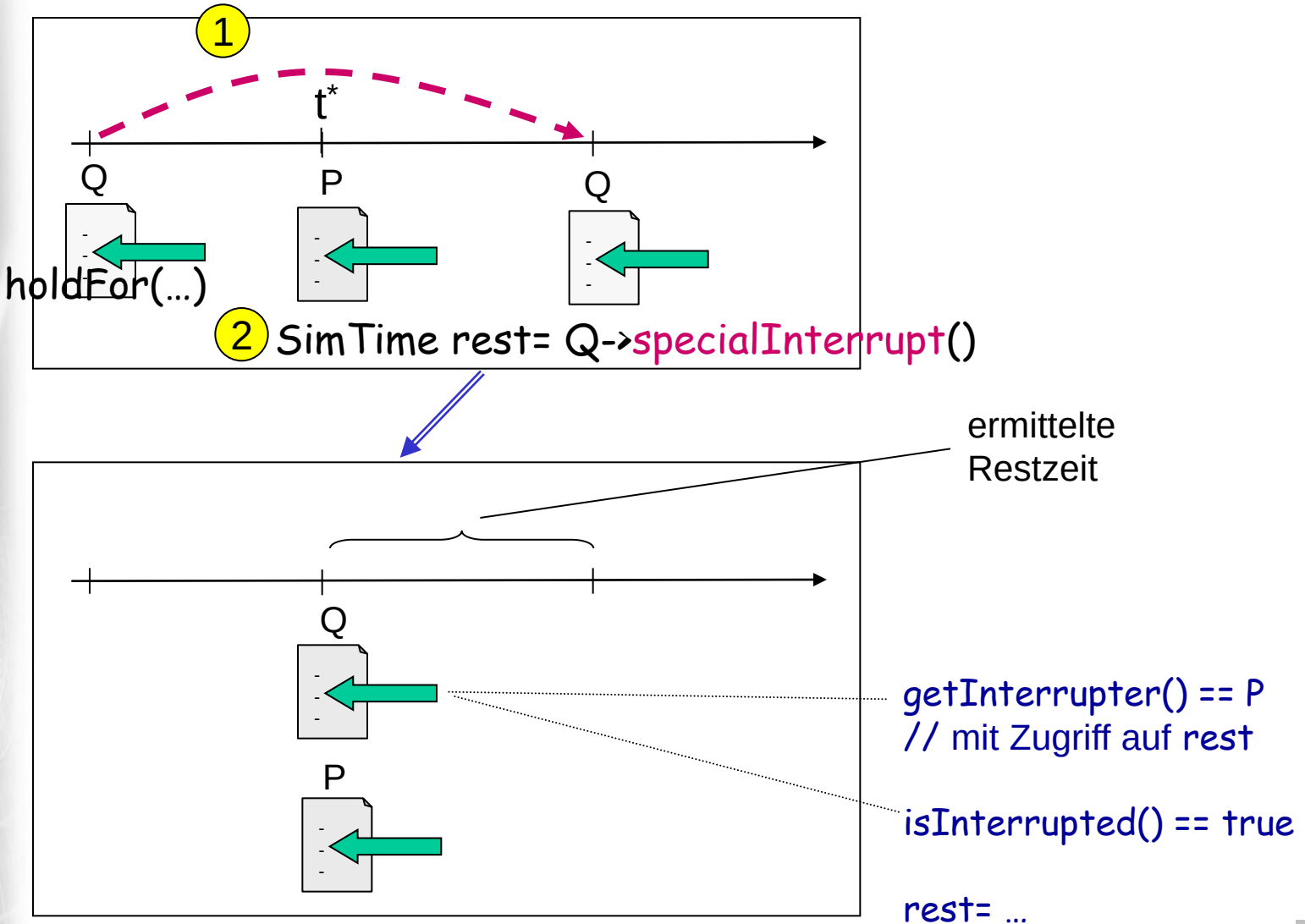
Nutzer-Kaskadierung von interrupt könnte die Restzeit ermitteln:

*geplante Ereigniszeit
der Beendigung einer Aktivität des
zu unterbrechenden
Prozesses*

```
SimTime specialInterrupt() {
        SimTime t= getExecutionTime() – getSimulation()->getTime();
        interrupt();
        return t;
}
```

*Unterbrechungszeitpunkt,
(aktuelle Modellzeit des interrupt-Rufers)*

Interrupt (detaillierter)



Interrupt (detaillierter)

Unterbrechungsbehandlung

```
bool isInterrupted() const {return interrupted;}
            // Abfrage eines Interrupt-Zustandes (nach erfolgtem interrupt)
            // true, falls Unterbrechung erfolgte und noch keine Verzögerung
            // stattgefunden hat


Sched* getInterrupter() const {return interrupter;}
            // Anzeige des Prozesses/Ereignisses, der/das interrupt() gerufen hat
            // falls isInterrupted() == true und
            //        getInterrupter()==0: dann war Interrupter der
            //  Simulationskontext


void resetInterrupt() {interrupted=false; interrupter=0;}
            // löscht Interrupt-Zustandseinträge
            // implizit bei jeder Scheduling-Operation
```

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionalität
5. Prozesswarteschlangen (ProcessQueue, Queue)
6. Spezielles Process-Scheduling (Memory)

Process: Lebenslauf und Rückgabewert

Process-Verhaltensfunktion

```
protected:
    virtual int main() = 0;
```

… bei Terminierung mittels return

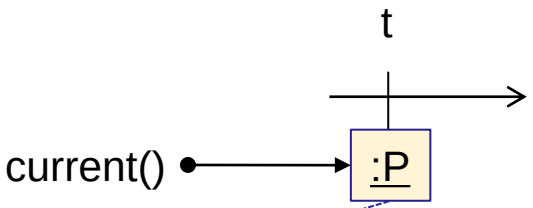
```
bool hasReturned() const {return validReturn;};
            // Test auf Beendigung

int getReturnValue() const;
            // liefert Rückgabewert (ohne evtl. Blockierung des Rufers)
            // Warnung für Nutzung eines ungültigen Wertes
            // vorheriger Test mit hasReturned()
```

Zustandsbedingte Blockierung/De-blockierung

Rückgabewert

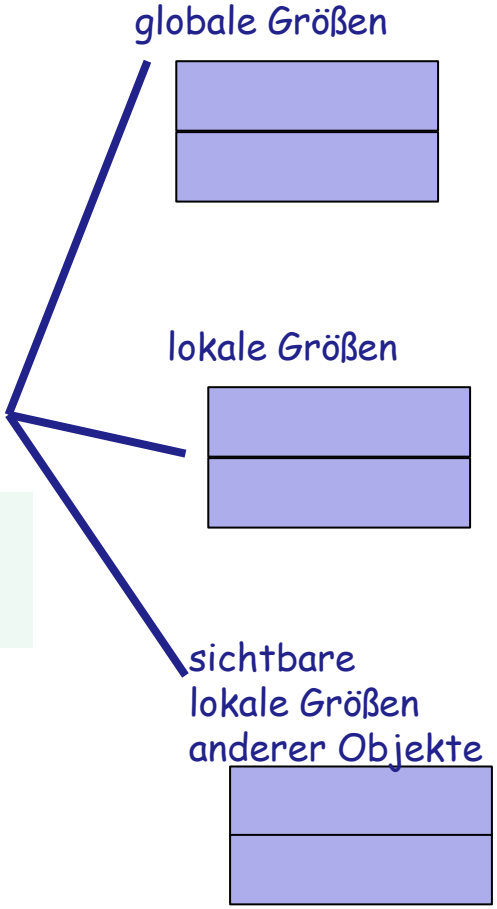
TRUE: Warten wurde mit interrupt durch anderen Prozess abgebrochen (Fortsetzung, obwohl Bedingung nicht erfüllt ist)

FALSE: sonst (also Bedingung erfüllt)




```
…
bool result= waituntil (special_condition);
//Fortsetzung g.d.w. Bedingung erfüllt ist
…
```

Boolscher Ausdruck über..



Synchronität
von Bedingungserfüllung und Deblockierung

Bedingung erfüllt
unmittelbare Fortsetzung der Ausführung

Bedingung nicht erfüllt
- Blockierung (→IDLE)
- Aktivierung bei Wertänderung einer der beteiligten Größen
- erneute Ausdrucksberechnung

PROBLEM: wie ist der Parameter von waituntil anzugeben ?

Lösung

(1) historisch: Namensparameterübergabe in Algol 60, Simula 67

```
function waituntil (name boolean cond) return boolean {
        while (not cond) do {
                sleep();
                if interrupted() return true;
        }
        return false;
}
```

- Aktueller Parameter wird bei Übergabe nicht berechnet,
- vielmehr bleibt der Ausdruck erhalten und
- wird an alle Aufrufstellen innerhalb des Funktionskörpers kopiert

(2) C++:

- Zustandsbedingung kein Boolscher Ausdruck, sondern eine Boolsche **Funktion**
- cond-Parameter von waituntil wird ein **Funktionszeiger** mit **Signatur-Constraint** (dieses wird per **Funktionstyp** festgelegt)
- die gewünschte Boolsche Funktion muss dann selbstverständlich von diesem Typ sein

Vordefinierte Signaturen für Bedingungsfunktionen in ODEMx

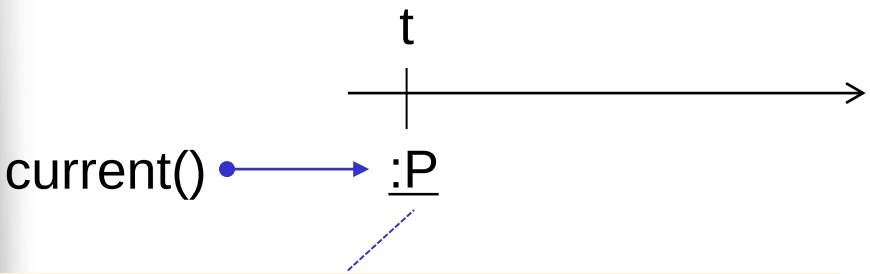
```
// Definierte Funktionstypen in ODEMx

typedef  bool (Process::*Selection)(Process* partner);

typedef  bool (Process::*Condition)();

typedef  double (Process::*Weight)(Process* partner);
```

Rückgabewert: bool

1. (impliziter) Parametertyp, der this-Zeiger-Typ: Process
2. weitere Parameter: keine

Mit dem Typ ist in C++ nur die Signatur definiert, das Verhalten ist völlig beliebig

Prinzipielle Anwendung




```
…
bool result= waituntil (&special_condition);
… //Fortsetzung g.d.w. condition erfüllt
```

```
class P:   public Process {
public:
…
        bool special_condition() {
            return ((x || y) && z)
        }
        int main ()
}
```

konform zu

```
typedef  bool (Process::*Condition)();
```

Zeiger zu einer Funktion

Implementation in ODEMx

```
bool waituntil( base::Condition cond ) {
            while (not (this->*cond) () ) do {
                        sleep();
                        if interrupted() return true;
            }
            return false;
}
```

Klassendefinition (Auszug)

Private Member-Variablen

```
private:
        ProcessState processState;  //process state
        Priority p;                 // process priority
        SimTime t;                  // process execution time
        Simulation* env;            // process simulation
        int returnValue;            // return value of main()
        ProcessQueue* q;            // pointer to queue if process is waiting
        SimTime q_int;              // enqueue-time
        SimTime q_out;              // dequeue-time
        bool validReturn;           // return value is valid
        bool interrupted;           // Process was interrupted
        Process* interrupter;       // Process was interrupted by
                                    // interrupter
                                    // (0 -> by Simulationkontext)
```

3. Prozessverwaltung

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: (ProcessQueue, Queue)
6. Spezielles Process-Scheduling (Memory)

Lokalisierung eines Prozesses (Überblick)

ein beliebiger Prozess kann zu einem Zeitpunkt
gleichzeitig in verschiedenen Listen erfasst sein:

- in höchstem einem Terminkalender (ExL)
  (seines Simulationskontextes, falls aktiviert)

- in höchstens einem ProcessQueue -Objekt
  (oder Ableitung davon)

- Modul Synchronistation definiert Queue als ProcessQueue-Ableitung,
- ProcessQueue-Objekte
  werden insbesondere zur Erfassung (inkl. Statistik) von blockierten Prozessen
  in vordefinierten Synchronisationsklassen (Bin, Res, Waitq, Condq, …) benutzt

- in beliebig vielen Memory-Listen der Typen (oder Ableitungen)
  - PortTail,
  - PortHead,
  - Timer,
  - WaitCondition

ProcessQueue

- **sortierte Liste** von Prozessen (Zeigern)
  in Form nutzergesteuerter Halbordnungen
  - DefaultOrder: sortiert nach Ausführungszeiten
    (u. bei Gleichzeitigkeit nach Priorität)
  - PriorityOrder: sortiert nur nach Priorität

  – zusätzlicher LIFO/FIFO- Auswahlparameter
  (ähnlich zum Konzept der ExL)

- **Anwendung**
  – Spezialisierung zur Klasse Queue
  im Modul Synchronisation zur Verwaltung schlafender/blockierter Prozesse (IDLE-Zustand)
  Führung einer Statistik über die Queue-Benutzung
  – nutzereigene Listen: nächstes Beispiel

- **Funktionen** (Varianten) zur Reaktivierung blockierter Prozesse
  - awake

ProcessQueue

Member-Funktionen

```
// Zugriffsmethoden
        Process* getTop() const;
        const std::list<Process*>& getList() const;

        bool isEmpty() const;

        unsigned int getLength() const {return (unsigned int)l.size();}

// Manipulationsmethoden
        virtual void popQueue(); // entfernt getTop()
        virtual void remove(Process* p);
        virtual void inSort(Process* p,  bool fifo = true);
```

ProcessQueue

- Nochmal: Ein Prozess kann zu einem Zeitpunkt immer nur in einer ProcessQueue enthalten sein

- Fehlbenutzung
  - bei Eintrag in zweite ProcessQueue:
    (ohne aus der ersten entfernt worden zu sein)
    Abbruch mit Fehlermeldung

- Prioritätsänderung
  - verursacht automatische Positionsänderung in der jeweiligen ProcessQueue

ProcessQueue

```
class ProcessQueue {
// Interface
    public:
     ProcessQueue(ProcessOrder* pred = &defOrder);
     Process* getTop() const;
     const std::list<Process*>& getList() const;

     bool isEmpty() const;
     unsigned int getLength() const {return (unsigned int)l.size();}

     virtual void popQueue();
     virtual void remove(Process* p);
     virtual void inSort(Process* p,  bool fifo = true);

// Implementation
    private:
     std::list<Process*> l;
     ProcessOrder* order;
};
```

&priOrder

```
void awakeAll(ProcessQueue* q);
void awakeFirst(ProcessQueue* q);
void awakeNext(ProcessQueue* q, Process* p);
```

Aktivierung
nach Current-Process
(kein Steuerungswechsel)

Process-Memberfunktionen

(Benutzung im Modul Synchronisation)

```
ProcessQueue* getQueue() const {return q;}
            // liefert Zeiger zur Warteschlange, in der sich der Prozess
            // befindet

SimTime getEnqueueTime() const {return q_int;}
            // liefert Eintrittszeit in die Warteschlange, in der sich der Prozess
            // befindet

SimTime getDequeueTime() const {return q_out;}
            // liefert Zeit des Verlassens der Warteschlange, in der sich der
            // Prozess befand
```

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: (ProcessQueue, Queue)
6. Spezielles Process-Scheduling (Memory)

Klasse Memory

Abstrakte Basisklasse für
- PortHeadT, PortHead,
- PortTailT, PortTail
- Timer,
- WaitCondition

```
class Memory {
public:
        Memory( base::Simulation& sim, const data::Label& label, Type type, MemoryObserver* obs = 0 );
        virtual ~Memory();

        virtual bool remember( base::Sched* newObject );
        virtual bool forget( base::Sched* rememberedObject );

        bool processIsWaiting (base::Process& process) const;
        virtual void eraseMemory();

        virtual bool isAvailable();
        bool waiting() const;

        SizeType countWaiting() const;

        virtual Type getMemoryType() const;

        virtual void alert();


privat:
        std::list< Sched * > memoryList //Liste vermerkter Sched-Objekte (Zeiger)
```

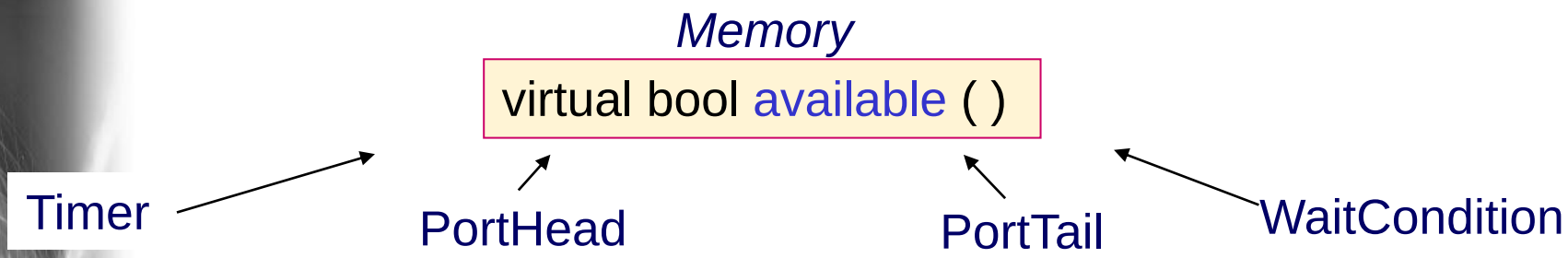
```
enum Type {
    TIMER,
    PORTHEAD,
    PORTTAIL,
    CONDITION,
    USERDEFINED
};
```

Spezialisierungen legen Semantik von isAvailable fest

Memory-Funktionalität

bei Alarmierung: alert()-Aufruf

- Aktivierung der blockierten Sched-Objekte zum aktuellen Zeitpunkt
- abermalige Ausführung der spezifischen available-Funktion
- Erneute Blockierung oder Ausführungsfortsetzung mit Verlassen der Memory-Liste



Timer	PortHead	PortTail	WaitCondition
virtual bool isAvailable( )	**virtual bool** isAvailable ( )	**virtual bool** isAvailable ( )	**virtual bool** isAvailable ( )
returns true if this timer is not scheduled	**returns** true if the buffer is not empty	**returns** true if the buffer is not full	**returns** condition-result
Timeout-Aktion beendet: blockierte Prozesse warten auf Zeitsignal (-verfügbarkeit)	blockierte Empfangs-prozesse warten auf Nachrichtenverfügbarkeit	blockierte Sende-prozesse warten auf Pufferverfügbarkeit	blockierte Prozesse warten auf Bedingungserfüllung

Motivation für eine komplexe wait-Funktion

typisches Synchronisationsproblem

- Prozesse (z.B. Zustandsmaschinen) setzen ihren Lebenslauf (Zustandsübergänge) nur unter bestimmten Bedingungen fort:
  - in einem von mehreren Eingangspuffern wurde eine Nachricht/Ereignis/Anforderung hinterlegt
  - eins von mehreren Zustandsereignissen ist eingetreten
  - eins von mehreren Zeitereignissen (ausgelöst durch Wecksignale von Uhren) sind eingetreten

- dabei kann genau ein Prozess oder aber es können auch mehrere betroffen sein, wobei wiederum eine selektive Auswahl der Prozesse modellierbar sein sollte

Lebenslauf eines Prozesses (Ausschnitt)

```
...
result= wait (buffer1, buffer2, timer1, cond1);
switch (result->getType() ) {
   case TIMER: ...
   case PORTHEAD: ...
   case CONDITION
   default: ...
}
...
```

- *Aufrufer registriert sich jeweils bei* buffer1, buffer2, timer1, cond1 *und blockiert gegebenenfalls*

- *bei diesen Objekten sollten sich weitere Prozesse/Ereignisse registrieren können*

- *Objekte sorgen für synchrone De-Blockierung*

Rückgabewert von wait

polymorphe Memory-Zeiger

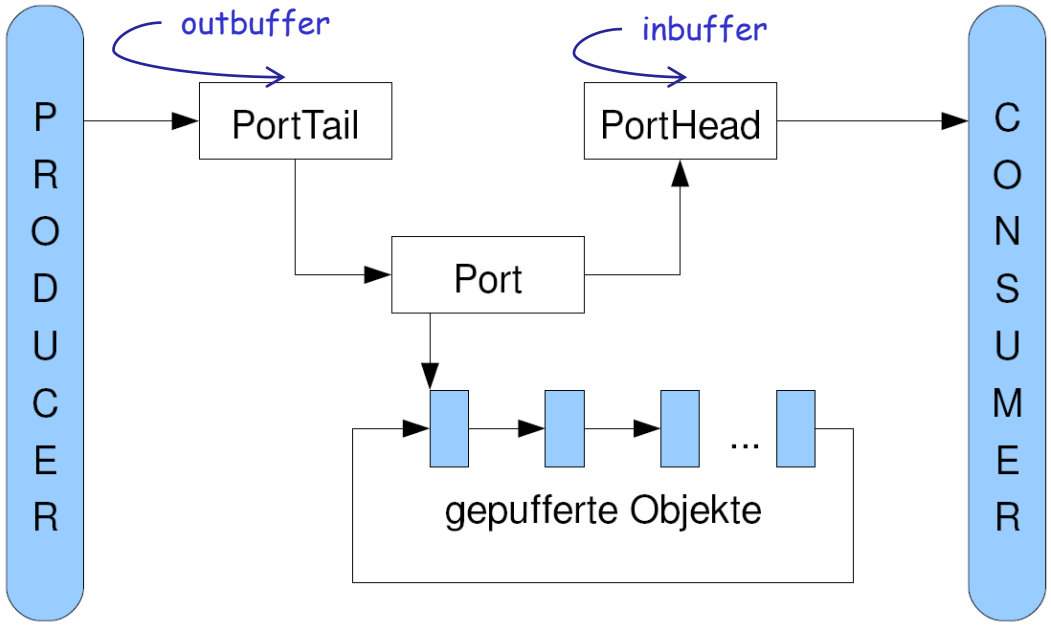
```
…
*Memory  m= wait (m1, m2, m3);
…
```

- Aufrufer-Prozess vermerkt sich in lokale Process*-Liste von m_1, m_2 und m_3 (falls deren „Verfügbarkeit“ nicht gegeben ist und blockiert
- wird auf den blockierten Prozess durch einen nebenläufigen Prozess ein **interrupt**() angewendet, verlässt dieser die m_1-, m_2- und m_3-Liste
- und beendet die **wait()**-Anweisung mit der Rückgabe des NULL-Zeigers (externe Unterbrechung der Blockierung)
- Sobald die „Verfügbarkeit“ von mindestens einem m_i gegeben ist, wird **wait** mit Rückgabe von m_i verlassen (der Prozess hat zuvor die lokalen Listen von m_1, m_2 und m_3 verlassen)
- Sollten mehr als zwei m_i‘s „Verfügbarkeit“ anzeigen, liefert **wait** den ersten von ihnen in der Parameterliste

Memory-Spezialisierungen: PortTail, PortHead

evtl.Blockierung des Aufrufers

```
...
outbuffer->put (...);
...
```



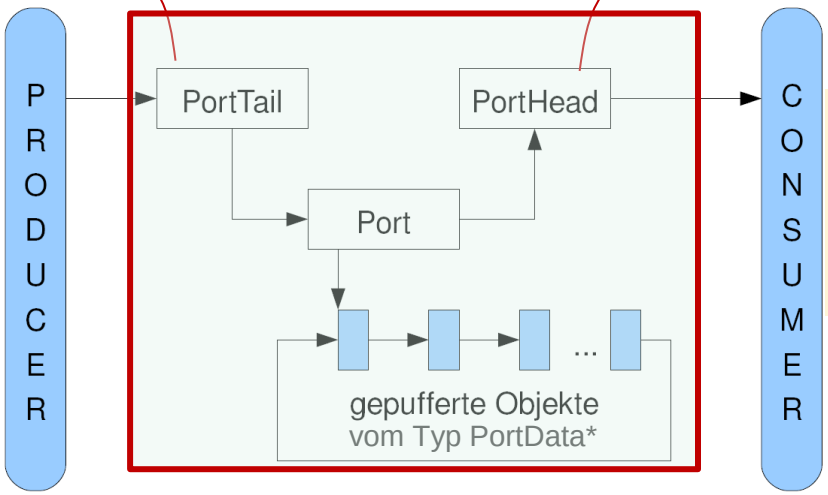

```
...
PortData*
inBuffer->get ()
...
```

Memory-Eigenschaft sichert die spezifische Blockierung der Aufrufer und ihre synchrone Aktivierung bei Aufhebung der Unterbrechungsbedingung.

Nachrichtenpuffer *zur Prozesskommunikation*

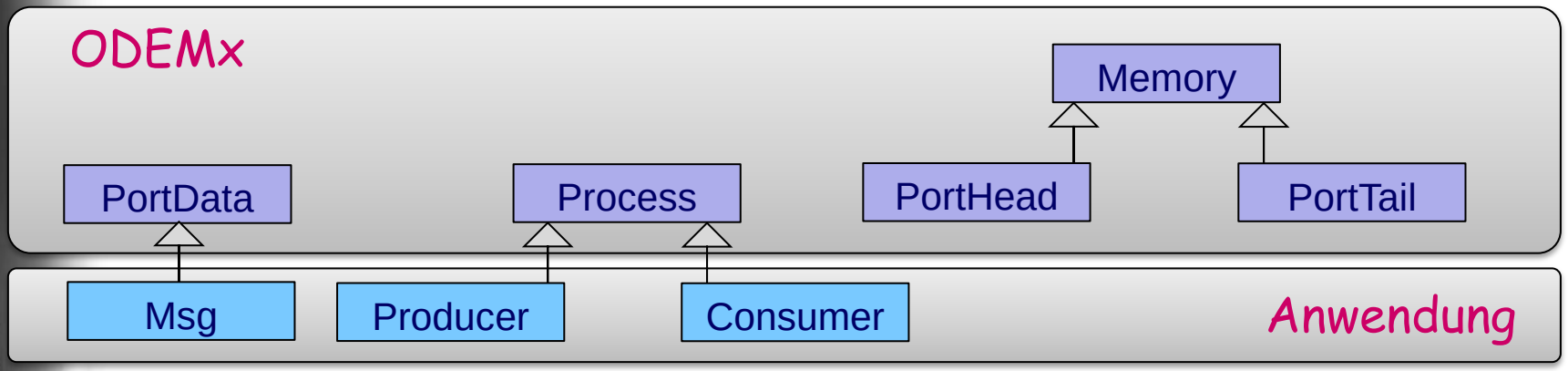
- Pufferverwaltung als Ensemble von Objekten der ODEMx-Klassen
  - Port Aufnahme der Nachrichten
  - PortTail Eingabeschnittstelle inkl. Memory für blockierte Producer-Prozesse im Fall eines voll belegten Ports
  - PortHead Entnahmeschnittstelle inkl. Memory für blockierte Consumer-Prozesse im Fall eines leeren Ports
  - PortData Basisklasse für Nachrichten

```
PortTail* outbuffer;
…
Msg* m= new Msg(…)
outbuffer->put (m);
…
```

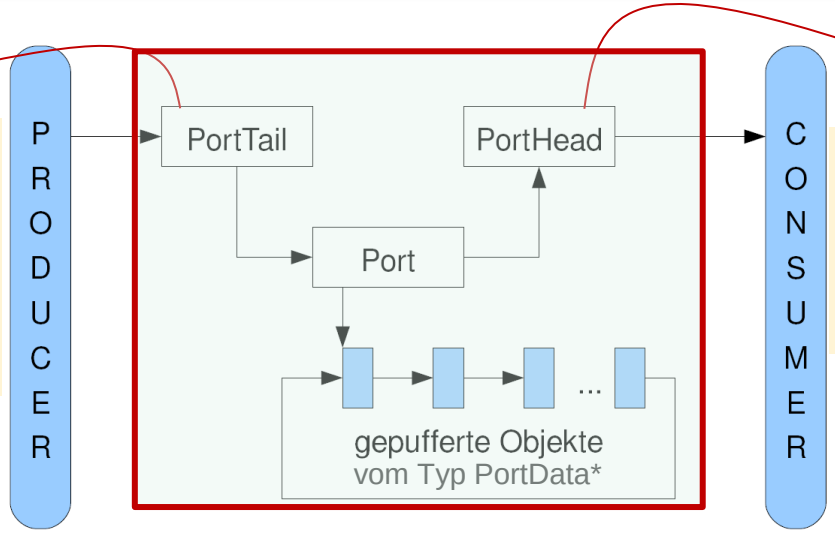



```
PortHead* inbuffer;
…
Msg* m= inBuffer->get();
…
```

PortTail, PortHead als Memory-Spezialisierungen




```
PortTail* outbuffer;
…
Msg* m= new Msg(…)
outbuffer->put (m);
…
```




```
PortHead* inbuffer;
…
Msg* m= inBuffer->get();
…
```

*Memory-Eigenschaft sichert die spezifische **Blockierung der Aufrufer** und ihre synchrone Aktivierung bei Aufhebung der Unterbrechungsbedingung.*

PortTail-, PortHead- Konstruktoren

PortHead (Simulation * *sim*,
Label *portName*,
PortMode *m* = WAITING_MODE,
unsigned int *max* = 10000)

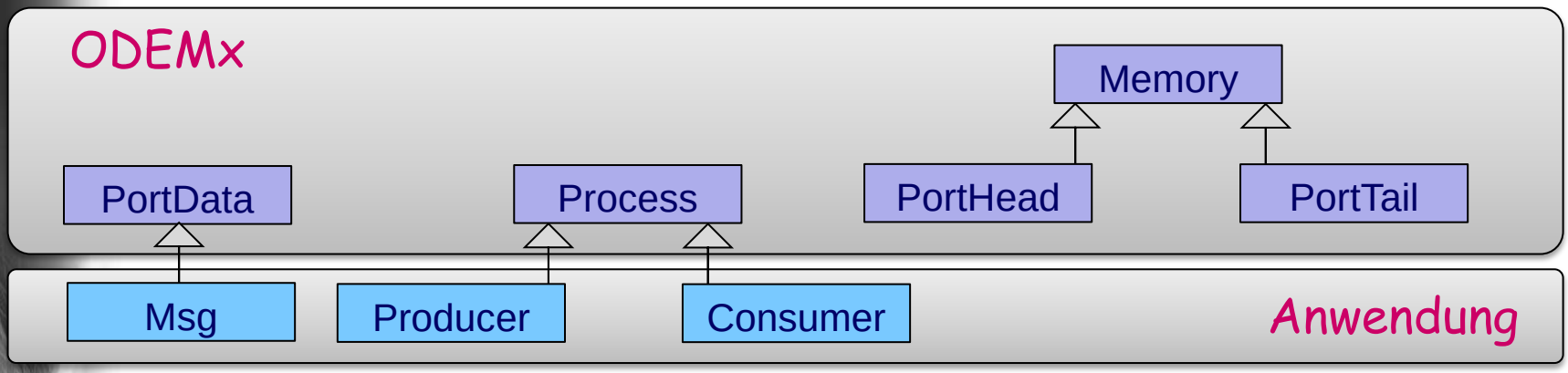
PortTail (Simulation * *sim*,
Label *portName*,
PortMode *m* = WAITING_MODE,
unsigned int *max* = 10000)

PortMode als Aufzählungstyp definiert

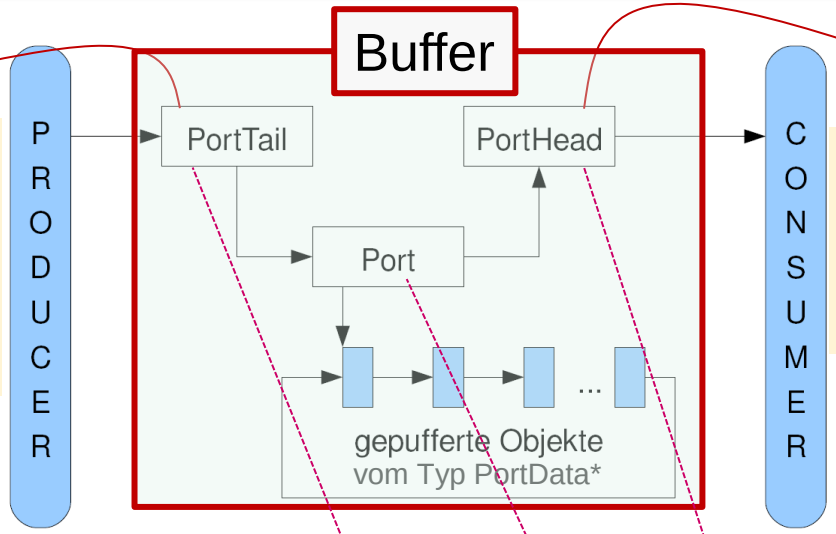
ERROR_MODE	Fehler, falls voll/leer
WAITING_MODE	Prozesswechsel, falls voll/leer
ZERO_MODE	Leeranweisung, falls voll/leer (0 als Return-Wert)

FRAGE: *Warum 3 Klassen statt 1 Klasse Buffer ?*

Unterstützung einer dynamischen Gestaltung von Verarbeitungsketten




```
Buffer* outbuffer;
…
Msg* m= new Msg(…)
outbuffer->put (m);
…
```




```
Buffer* inbuffer;
…
Msg* m= inBuffer->get();
…
```

Ensemble von drei Listen

blockierte Producer

blockierte Consumer

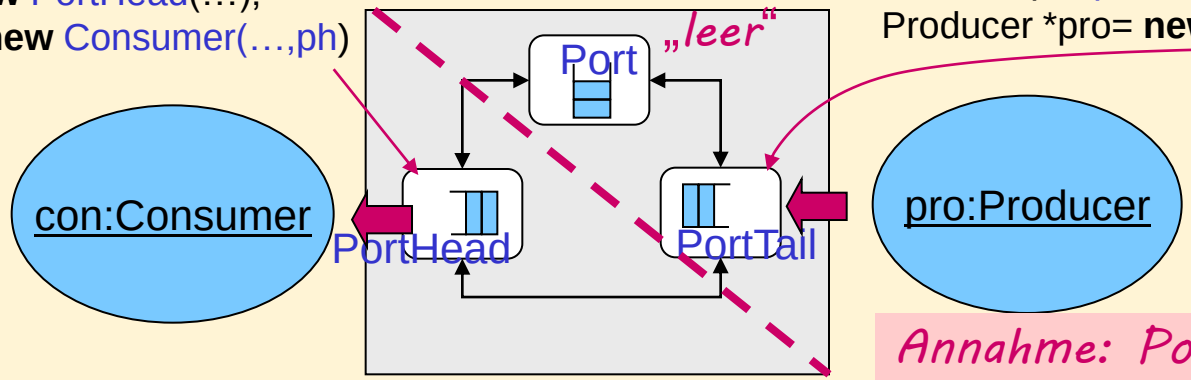
gepufferte Nachrichten

Weitere Port-Funktionalität: cut(), splice()

„vor cut()-Anwendung“

```
PortHead *ph= new PortHead(…);
Consumer *con= new Consumer(…,ph)
```

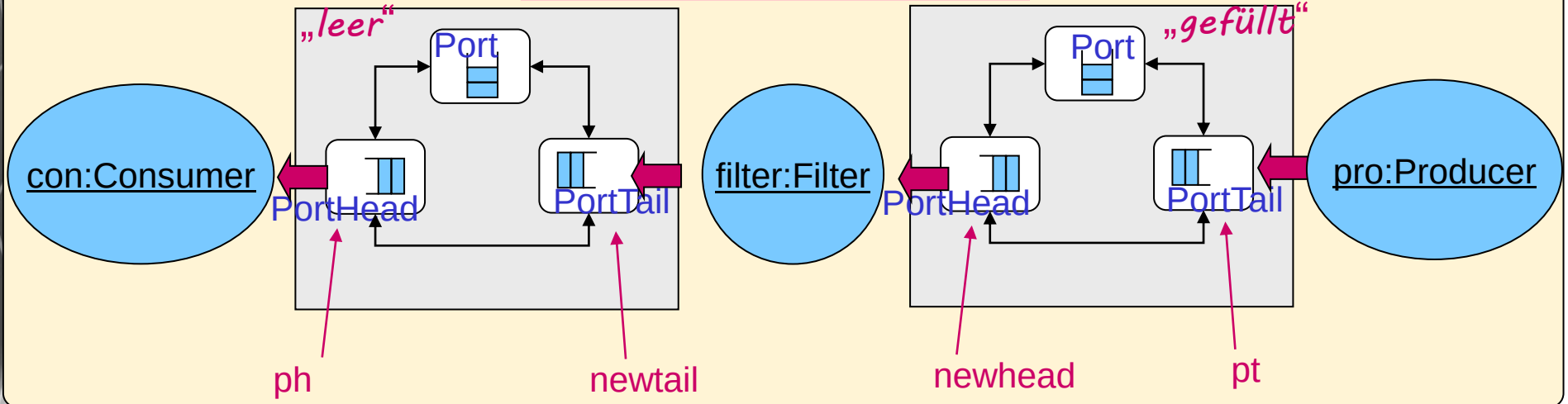
```
PortTail *pt= ph.tail()
Producer *pro= new Producer(…, pt)
```



Annahme: Port sei gefüllt

```
PortHead *newHead= ph->cut();  //bei Ergänzung eines neuen PortHead-Zugangs
PortTail *newTail= ph->tail(); // Ergänzung eines neuen PortTail-Zugangs mit leerem Port

Filter *filter= new Filter(…, newHead, newTail)
```

„nach cut()-Anwendung“



Zusammenfassung

- Process-Member-Funktion wait

```
Memo* wait (Memo* m0,
            Memo* m1= 0, Memo* m2= 0, Memo* m3= 0, Memo* m4= 0, Memo* m5= 0 )
```

liefert eines der Memo-Objekte zurück, sobald dieses „Verfügbarkeit“ liefert;
bis dahin bleibt der Aufrufer blockiert

- Beispiel

```
PortHead *p1, *p2, *p3. *p;
```

```
...
p= wait (p1, p2, p3);
...
```

Aufrufer-Prozess wartet(blockiert) bis in einem der Buffer p1, p2, p3
eine Nachricht hinterlegt worden ist

```
Memory *m;
PortHead *ph;
PortTail *pt;
Timer t;
```

```
...
m= wait (ph,pt, t);
switch (m->getMemoryType()) {
   case TIMER: ...
   case PORTHEAD: ...
   default: ...
}
```

Aufrufer-Prozess wartet(blockiert) bis in einem der Puffer ph eine Nachricht abgelegt wird **oder** in einem Puffer pt Platz geworden ist **oder** ein Timeout anliegt

Ein frohes
Weihnachtsfest
mit vielen,
aber nicht
zu vielen
Überraschungen

und kommen Sie gut ins Neue Jahr !