

Kurs OMSI ***im WiSe 2014/15***

Objektorientierte Simulation ***mit ODEMx***

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

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

3. *Prozess-Scheduling*

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionalität
5. Prozesswarteschlangen: ProcessQueue, Queue
6. Spezielle Process-Synchronisation: Memory, Port
7. Universelle wait-Anweisung

Varianten der Kontextaktivierung

Methoden der Klasse Simulation (Simulationskontext)
(aufgerufen vom C++ Hauptprogramm)

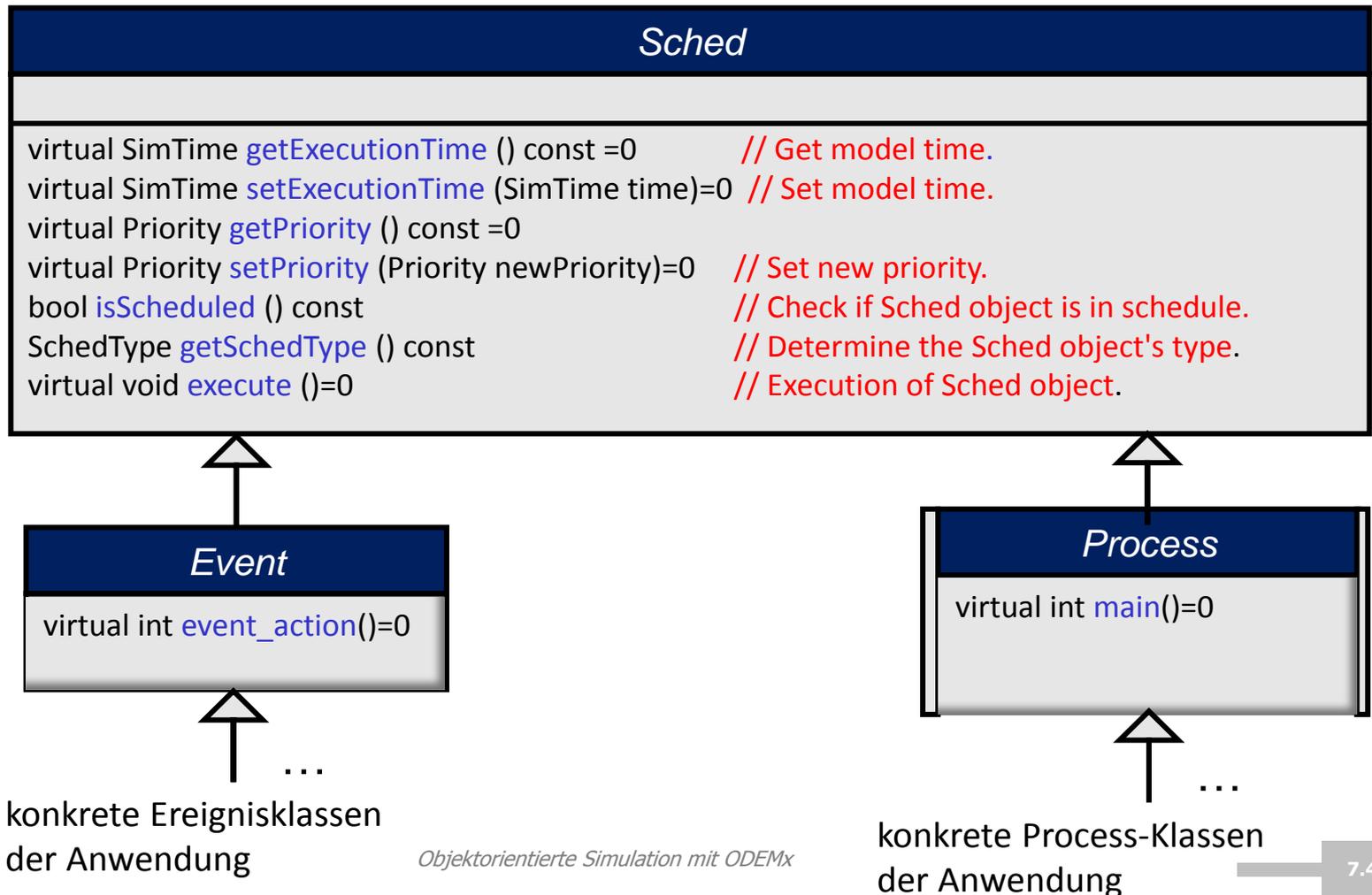
1. Einzelschrittausführung: `step()`
2. Lauf bis zum Erreichen/Überschreiten einer vorgegebenen Modellzeit (SimTime): `runUntil(...)`
3. Lauf bis zum Ende der Simulation: `run()`

Rückkehr ins C++ Hauptprogramm:

- **implizit:**
es gibt keinen **aktiven** Prozess mehr im zugehörigen Simulationskontext (Kalender ist leer)
- **explizit:** die Simulation wurde mit `exitSimulation()` durch einen Prozess des Simulationskontextes beendet

ExecutionList ~ Terminkalender

- jeder Simulationskontext verkörpert genau eine solche Liste von sortierten Sched-Zeigern (i.allg. Gemischte Liste von Ereignis- und Process-Objekten)
- der erste Eintrag hat die kleinste Ereigniszeit, höchste Priorität: er ist der Current-Prozess



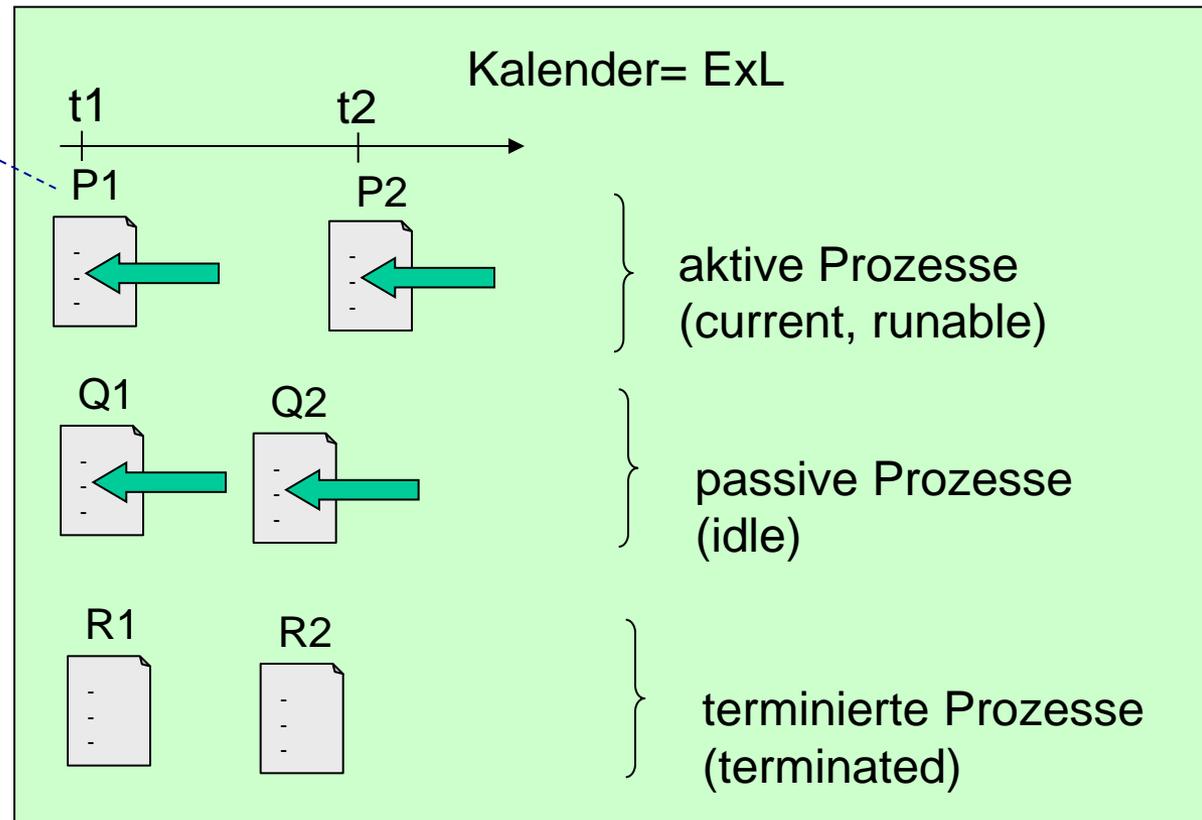
Elementare Prozessverwaltung in ODEMx

Klasse Process
verfügt
über eine virtuelle
Memberfunktion **main**
(Lebenslauf des Prozesses)

```
int main ( ... ) {  
...  
}
```

C++ main program

simulation context (DefaultSimulation-Objekt)



Ensemble von **main()**-Funktionen aller existenten
Prozess-Objekte wird zusammen mit eigentlichem C++
Hauptprogramm als Ensemble von Coroutinen
quasiparallel ausgeführt

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();
```

Blockierung,
Unterbrechung,
Beendigung

```
Priority getPriority() const;  
Priority setPriority( Priority newPriority );  
Priority getQueuePriority() const;  
Priority setQueuePriority( Priority newPriority, bool reactivate );
```

Lesen/Setzen von
Prioritäten

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

Infos zur
Ereigniszeit und
Unterbrechung

protected:

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

Lebenslauf

public:

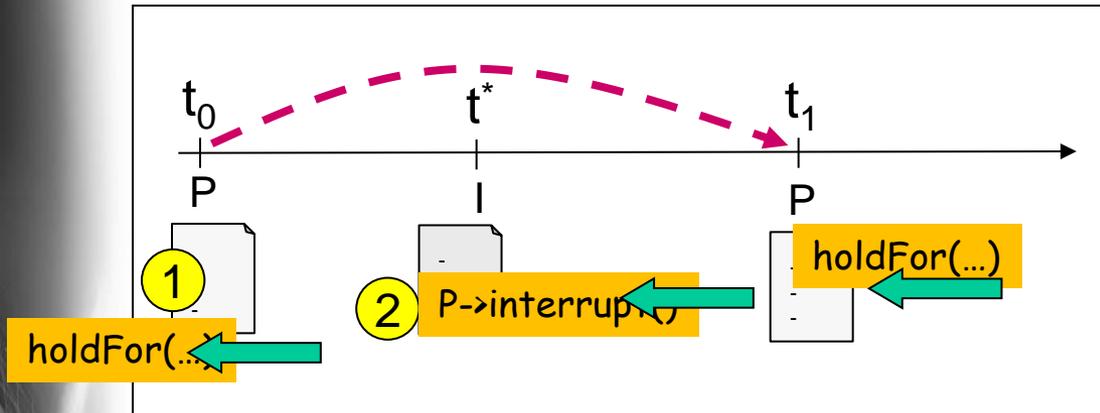
```
bool hasReturned() const {return validReturn;};  
int getReturnValue() const;
```

Infos zum
Lebenslauf-Ende

...

Process: Interrupt-Mechanismus (1)

anwendungsabhängige Zustandsänderung



1 P führt eine Aktion aus, die $(t_1 - t_0)$ ZE benötigt

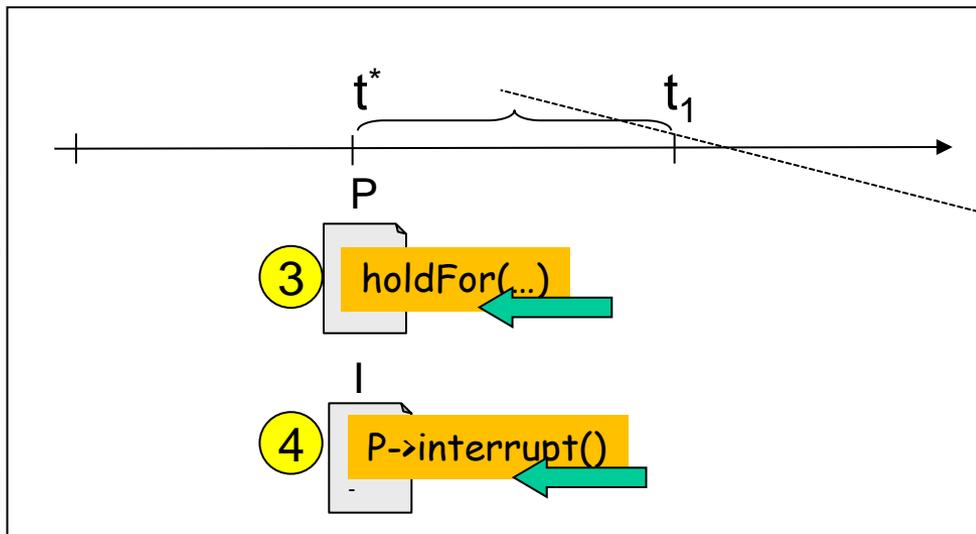
2 zum Zeitpunkt t^* wird P von I unterbrochen der geplante Endzeitpunkt für P ist nun hinfällig

P erhält zum Unterbrechungszeitpunkt die Steuerung zurück (vorab wurden Statusinfos in P hinterlegt)

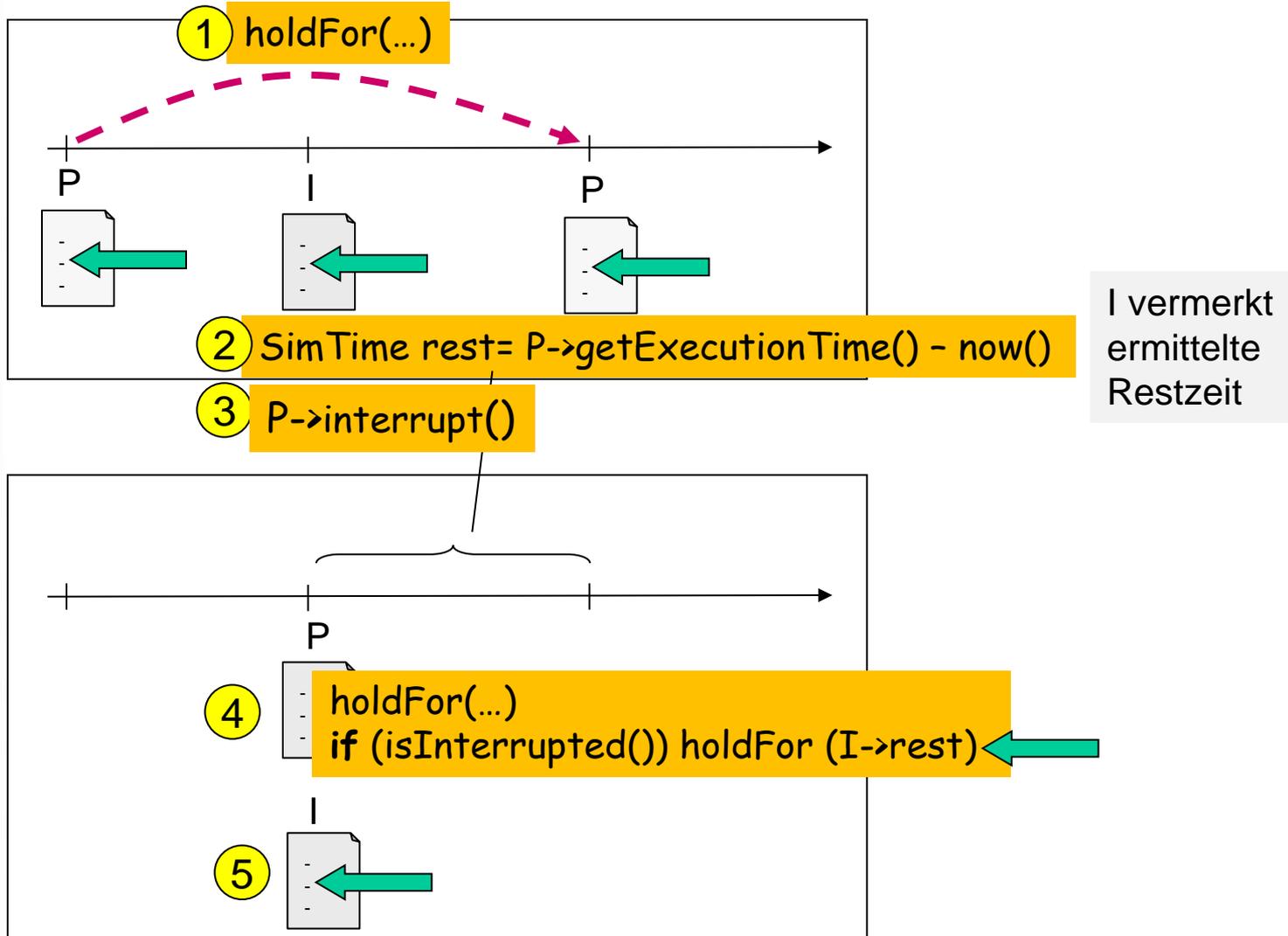
```
getInterrupter() == I  
// mit Zugriff auf rest
```

```
isInterrupted() == true
```

3 P kann Unterbrechungsbehandlung organisieren:



Process: Interrupt-Mechanismus (2)



Process: Interrupt-Mechanismus (3)

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 (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionalität
5. Prozesswarteschlangen: ProcessQueue, Queue
6. Spezielle Process-Synchronisation: Memory, Port
7. Universelle wait-Anweisung

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
```

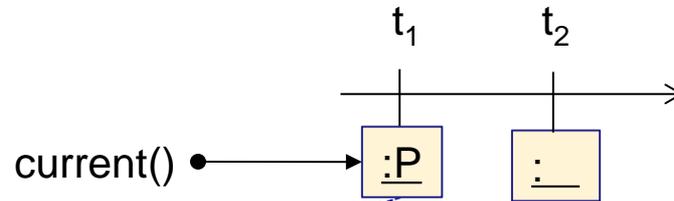
```
// vorheriger Test mit hasReturned()
```

Zustandsbedingte Blockierung/De-blockierung

Rückgabewert

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

TRUE: sonst (also Bedingung erfüllt)

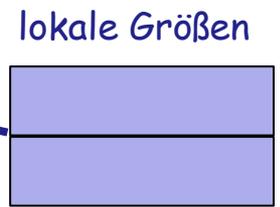
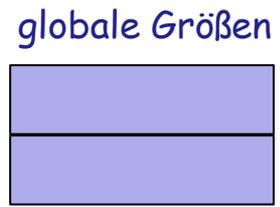


```

...
bool result= waituntil (special_condition);
...
    
```

unser Ziel: Synchronitätssicherung
von Bedingungserfüllung und Deblockierung

Boolscher Ausdruck über..



Bedingung erfüllt
unmittelbare Fortsetzung der Ausführung

Bedingung nicht erfüllt

- Blockierung per **sleep()**: CURRENT → IDLE
- Erneute 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 false;
    }
    return true;
}
```

- Aktueller Parameter wird bei Übergabe nicht berechnet,
- vielmehr bleibt Ausdrucksdefinition erhalten und
- wird an alle Aufrufstellen innerhalb des Funktionskörpers kopiert
- Aufrufer kann feststellen, ob waituntil per Zustandsereignis oder (externer) Unterbrechung verlassen worden ist

(2) C++:

- Zustandsbedingung kein Boolescher Ausdruck, sondern eine Boolesche **Funktion**
- `cond`-Parameter von `waituntil` wird ein **Funktionszeiger** mit **Signatur-Constraint** (Signatur wird per **Funktionsstyp** festgelegt)
- die gewünschte Boolesche Funktion (aktueller Parameter) muss dann selbstverständlich von diesem Typ sein

Vordefinierte Signaturen für Bedingungsfunktionen in ODEMx

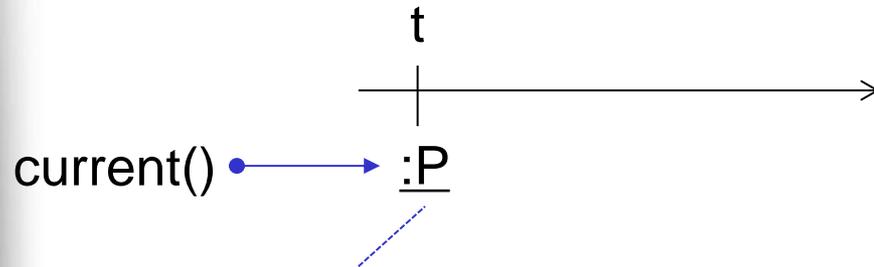
```
// Definierte Funktionstypen in ODEMx  
  
typedef bool (Process::*Selection)(Process* partner);  
  
typedef bool (Process::*Condition)();  
  
typedef double (Process::*Weight)(Process* partner);
```

Rückgabewert: **bool**

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

Mit der Typdefinition ist in C++ nur die Signatur festgelegt, 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 ()  
}
```

Implementation in ODEMX

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

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

konform zu

Zeiger zu einer Funktion

Fortsetzung mit **interrupt** von :P oder von einem Prozess per **activate**, der x, y oder z geändert hat

Klassendefinition (Auszug)

Private Member-Variablen

private:

```
ProcessState processState;    //process state
int priority;                // execution priority
int queue_priority;         // waiting 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-Funktionalität
5. Prozesswarteschlangen: ProcessQueue, Queue
6. Spezielle Process-Synchronisation: Memory, Port
7. Universelle wait-Anweisung

Lokalisierung eines Prozesses (Überblick)

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

- in höchstem einem Terminkalender
(seines Simulationskontextes, falls aktiviert)
- in höchstens einem Warteschlangen-Objekt (*ProcessQueue*)
(oder Ableitung davon)

- Modul *Synchronisation* definiert *Queue* als *ProcessQueue*-Ableitung,
- *Queue*-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)

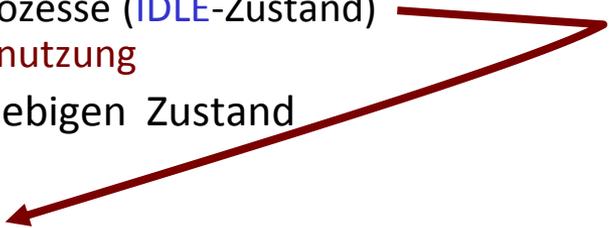
- *PortTail*,
- *PortHead*,
- *Timer*,

ohne weitere Ableitung nutzbar

- *WaitCondition*

Ableitung zwingend bei Festlegung der Memberfunktion

Motivation von ProcessQueue (bzw. Queue)

- **sortierte Liste** von Prozessen (Zeigern) im beliebigen Zustand
 - **DefaultOrder**: sortiert nach Ausführungszeiten (u. bei Gleichzeitigkeit nach Priorität)
 - **PriorityOrder**: sortiert nur nach Priorität
 - zusätzlicher LIFO/FIFO- Auswahlparameter
 - **Anwendung**
 - Spezialisierung zur Klasse **Queue** im Modul Synchronisation
 - zur Verwaltung schlafender/blockierter Prozesse (**IDLE**-Zustand)
 - Führung einer Statistik über die Queue-Benutzung**
 - nutzereigene Listen von Prozessen im beliebigen Zustand
 - **Funktion** zur Reaktivierung blockierter Prozesse
 - **awake** (in verschiedenen Suffix-Varianten)
- 

ProcessQueue

```
class ProcessQueue {  
// Interface  
public:  
    ProcessQueue ();  
    Process* getTop () const;  
    const std::list<Process*>& getList () const;  
  
    bool isEmpty() const;  
    unsigned int getLength() const {return (unsigned int) l.size();}  
  
    virtual void popQueue (); // entfernt getTop()  
    virtual void remove (Process* p);  
    virtual void inSort (Process* p, bool fifo = true); // QueuePriority  
  
// Implementation  
private:  
    std::list<Process*> l;  
    ProcessOrder* order;  
};
```

Eintrag nach Wartepriorität

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

ProcessQueue

- Ein Prozess kann zu einem Zeitpunkt immer nur in einer **ProcessQueue / Queue** enthalten sein
 - Fehlbenutzung
bei Eintrag in zweite **Warteschlange**
(ohne aus der ersten entfernt worden zu sein)
Abbruch mit Fehlermeldung
- **Prioritätsänderung**
 - verursacht automatische Positionsänderung
in der jeweiligen **ProcessQueue / Queue**

Weitere Process-Memberfunktionen

(mit Benutzung im Modul Synchronisation)

```
ProcessQueue* getQueue() const {return q;}  
    // liefert Zeiger zur Warteschlange, in der sich der Prozess  
    // befindet  
  
SimTime getEnqueueTime() const {return q_in;}  
    // 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-Funktionalität
5. Prozesswarteschlangen: ProcessQueue, Queue
6. Spezielle Process-Synchronisation: Memory, Port
7. Universelle wait-Anweisung
8. Beispiel: Autofähre

Klasse Memory

- PortHead,
- PortTail
- Timer,
- WaitCondition
- ... nutzereigene Klassen

~ dient als abstrakte Basisklasse zur Erfassung von blockierten Sched-Objekten für verschiedene Synchronisationskonzepte in ODEMX

```
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();
};
```

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

Spezialisierungen legen Semantik von `isAvailable` fest

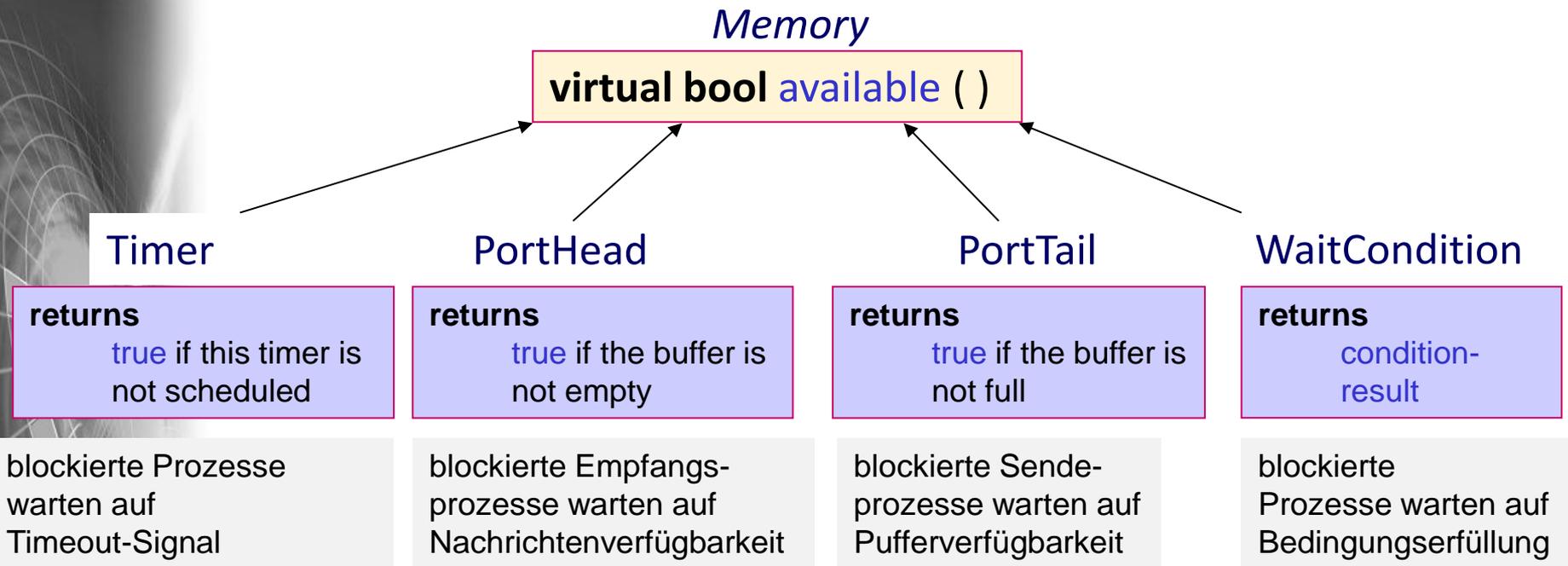
privat:

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

Memory-Funktionalität

bei Alarmierung: `alert()`-Aufruf

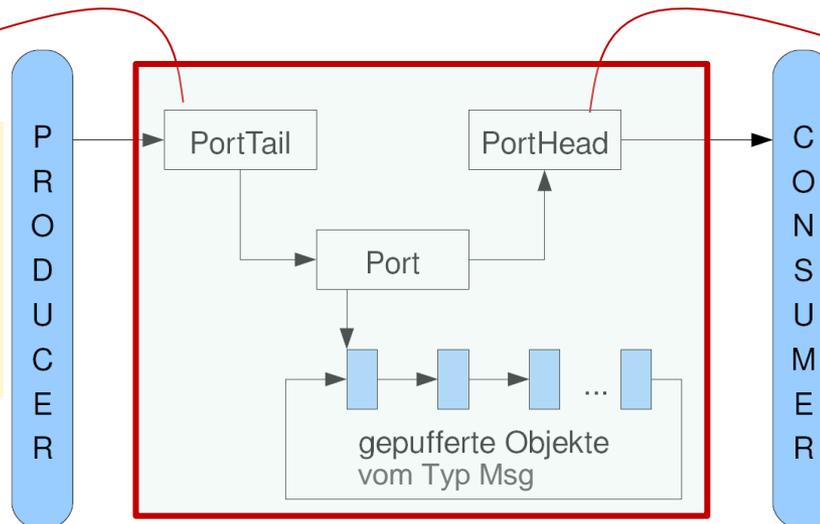
- Aktivierung der blockierten `Sched`-Objekte zum aktuellen Zeitpunkt
- abermalige Ausführung der spezifischen `available`-Funktion
- erneute Blockierung oder Ausführungsfortsetzung mit Verlassen der `Memory`-Liste



Nachrichtenpuffer zur Prozesskommunikation

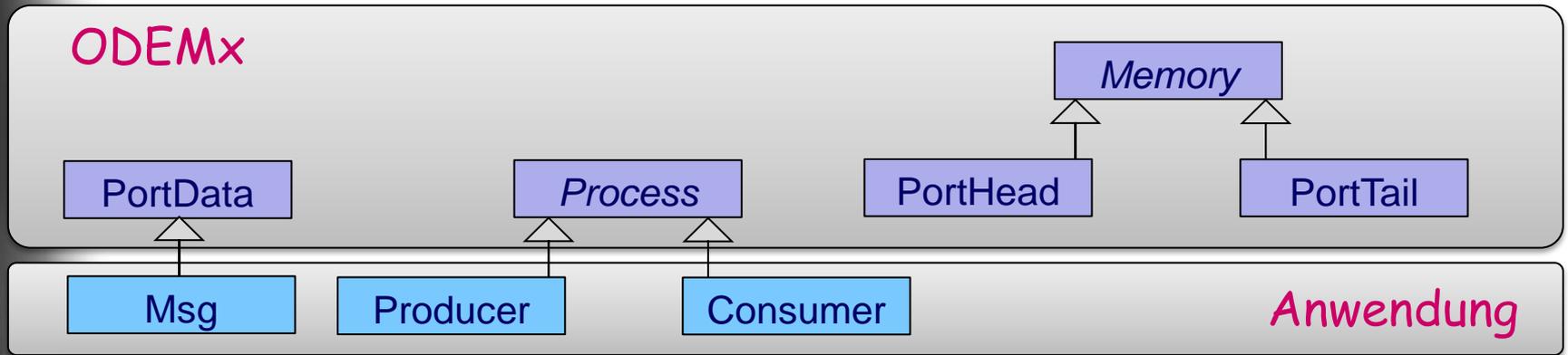
- Pufferverwaltung als Ensemble von Objekten der ODEMX-Klassen
 - **Port** (keine Memory-Ableitung) Aufnahme der Nachrichten (**PortData**-Objekte)
 - **PortTail** Eingabeschnittstelle inkl. Memory-Eigenschaft für blockierte Producer-Prozesse im Fall eines voll belegten Ports
 - **PortHead** Entnahmeschnittstelle inkl. Memory-Eigenschaft für blockierte Consumer-Prozesse im Fall eines leeren Ports
 - **PortData** Basisklasse für Msg

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



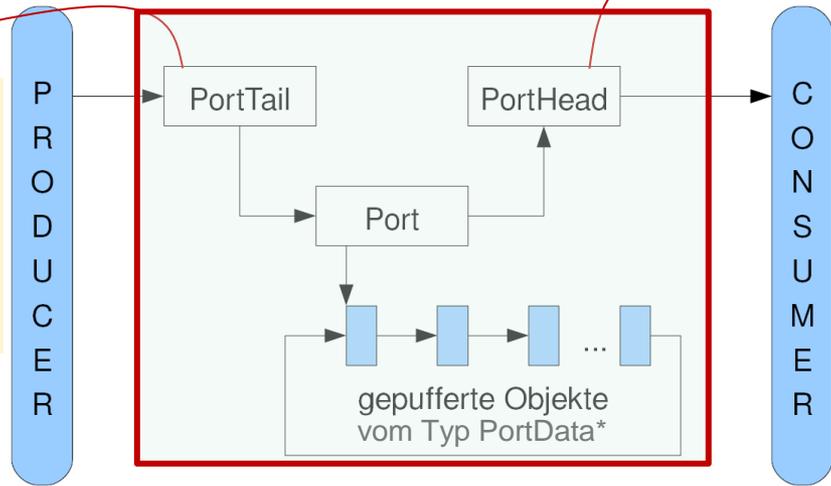
```
PortHead* inbuffer;  
...  
Msg* m= inBuffer->get();  
...
```

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

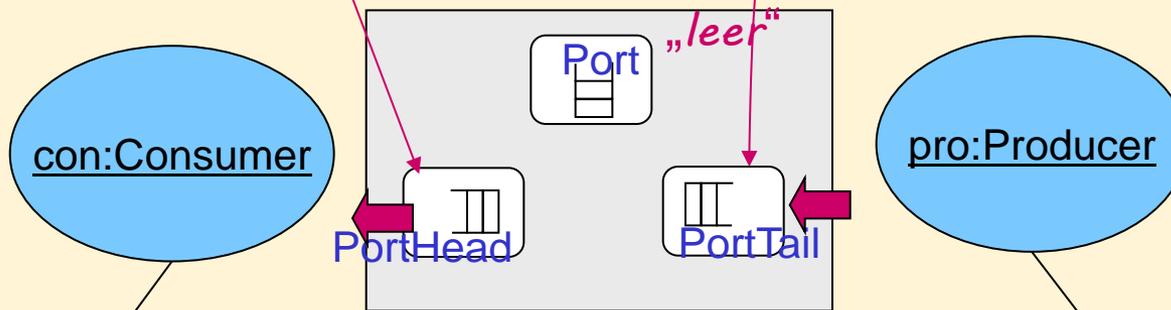
ERROR_MODE	Fehler, falls voll/leer
WAITING_MODE	Prozesswechsel, falls voll/leer
ZERO_MODE	Leeranweisung, falls voll/leer (0 als Return-Wert)

Anwendung:

C++ Hauptprogramm

```
PortHead *ph= new PortHead(...); // Entnahme  
Consumer *con= new Consumer(..., ph);
```

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



```
con->activate();
```

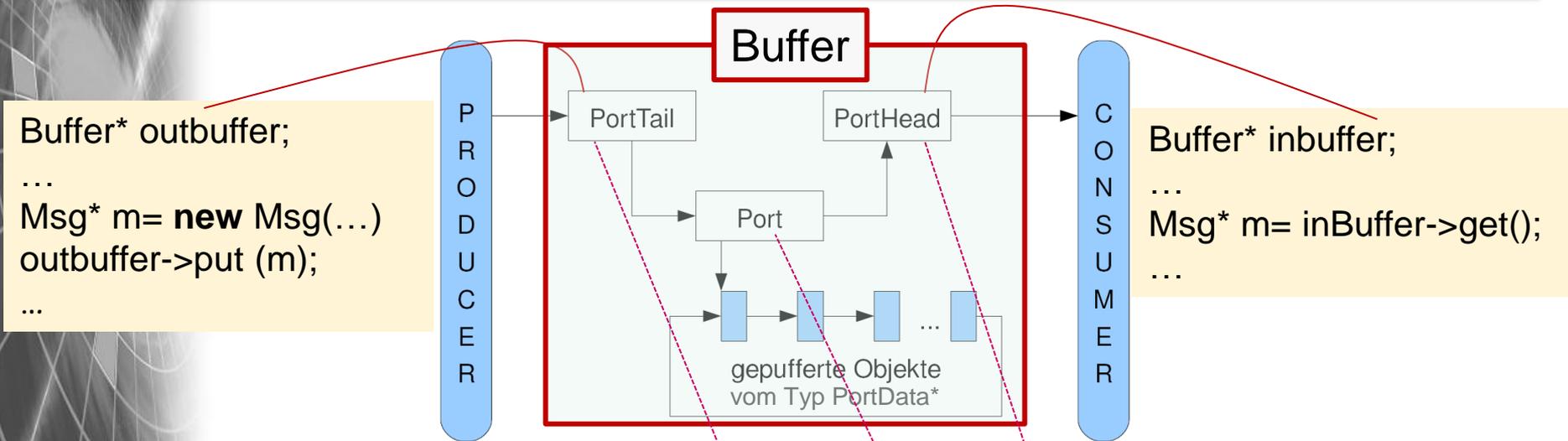
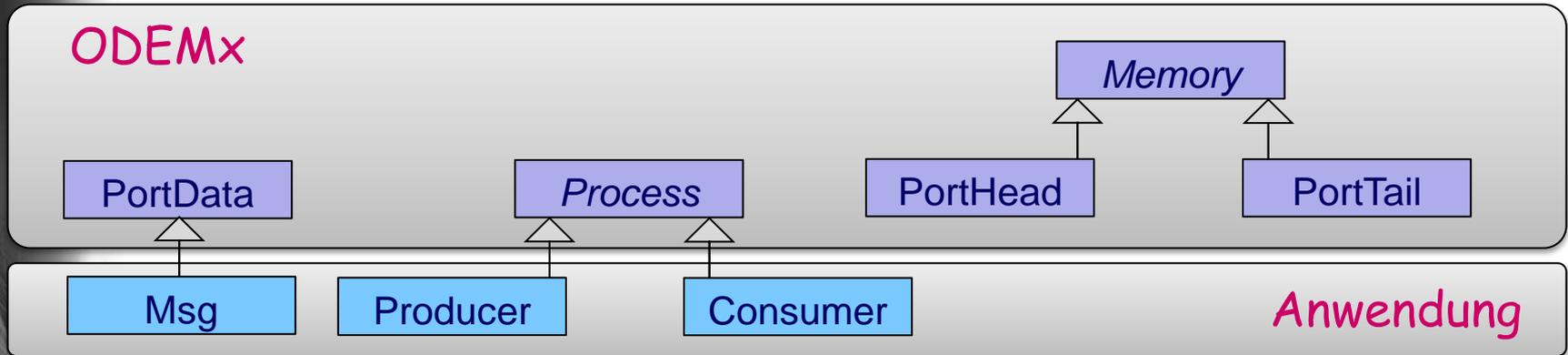
```
main() {  
  ...  
  Msg* m= ph->get();  
  ...  
}
```

```
main() {  
  ...  
  Msg* m= new Msg(...)  
  pt->put (m);  
  ...  
}
```

put/get-Semantik abhängig vom eingerichteten Modus

FRAGE: Warum 3 Klassen statt 1 Klasse Buffer ?

- (1) Memory-Eigenschaft (zwei verschiedene Fortsetzungsbedingungen)
- (2) Unterstützung einer dynamischen Gestaltung von Verarbeitungsketten



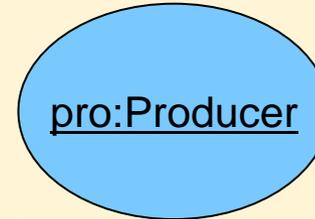
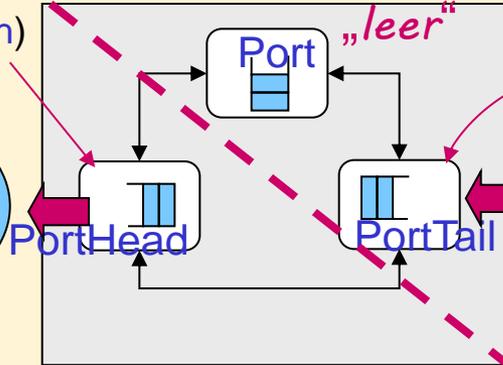
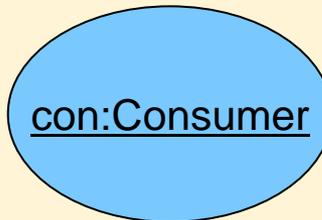
Ensemble von drei Objekten zur Verwaltung einer der drei Listen
 blockierte Producer blockierte Consumer
 gepufferte Nachrichten

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

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

„vor cut()-Anwendung“

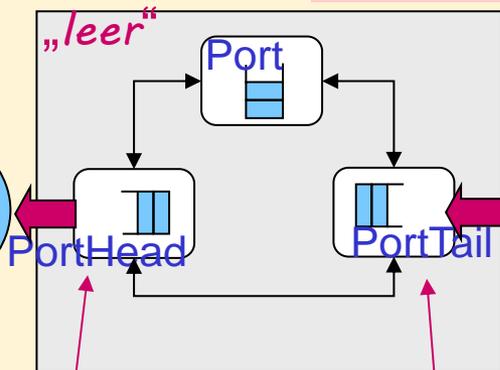
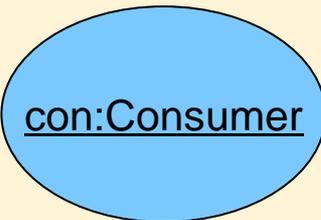
```
PortTail *pt= ph.tail()  
Producer *pro= new Producer(..., pt)
```



Annahme: Port sei gefüllt

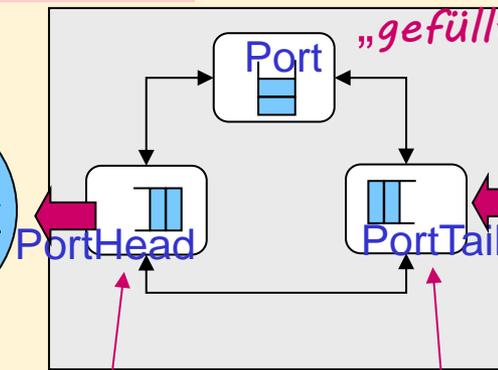
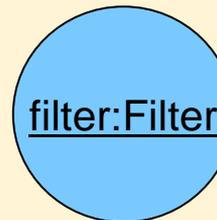
```
PortHead *newHead= ph->cut(); //bei Ergänzung eines neuen PortHead-Zugangs  
PortTail *newTail= ph->tail(); // Ergänzung eines neuen PortTail-Zugangs mit leerem Port  
Filter *filter= new Filter(..., newHead, newTail)
```

„nach cut()-Anwendung“



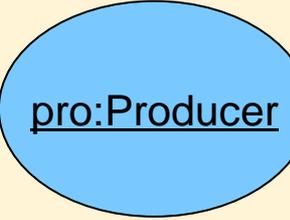
ph

newtail



newhead

pt



3. *Prozess-Scheduling*

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionalität
5. Prozesswarteschlangen: ProcessQueue, Queue
6. Spezielle Process-Synchronisation: Memory, Port
7. Universelle wait-Anweisung
8. Beispiel: Autofähre

Motivation für eine universelle 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 möglichen Zustandsereignissen ist eingetreten
 - eins von mehreren Zeitereignissen (ausgelöst durch Wecksignale von Uhren) ist eingetreten
- dabei kann genau ein Prozess betroffen sein oder mehrere

Lebenslauf
eines
Prozesses
(Ausschnitt)

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

- *wait*-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 parallelen 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 (nach Position in der Parameterliste)

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