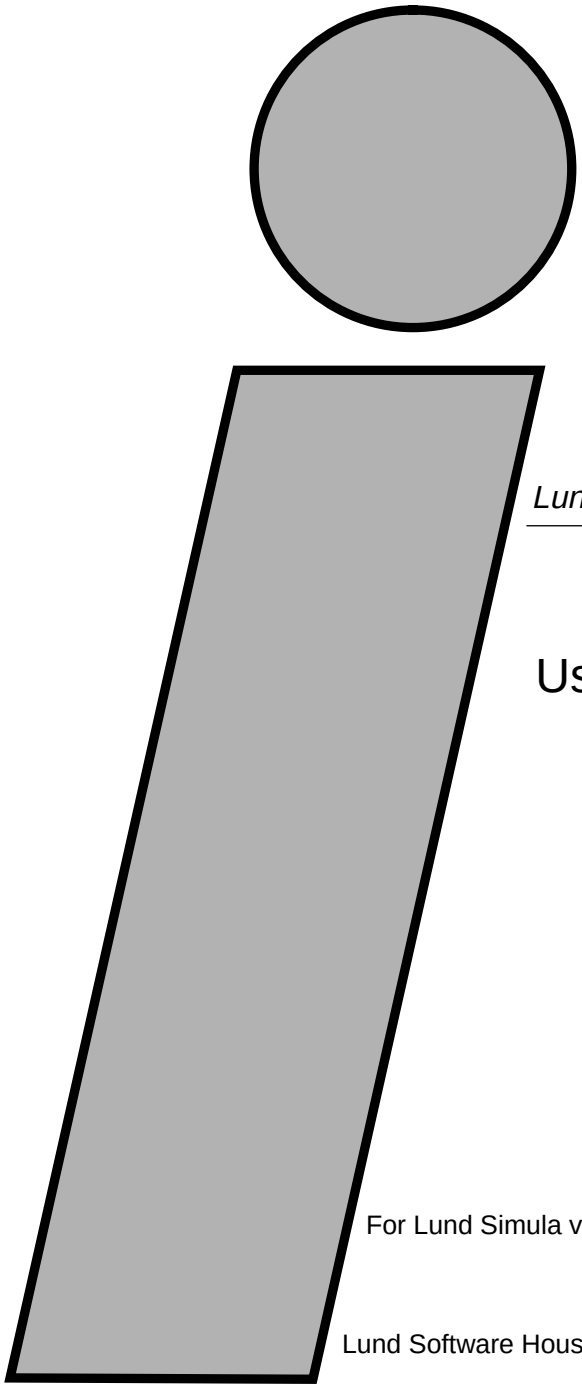


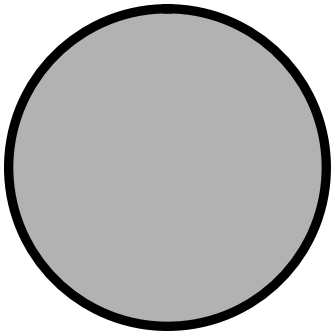
LUND

Lund Simula Documentation

Using the **simioprocess** library on Unix Systems

For Lund Simula version 4.15 or later

Lund Software House AB, Sweden



SIMULA

Lund Simula Documentation

Using Lund Simula on Unix Systems
Version 4.15

by Boris Magnusson and Per Holm

Printed at: 5 December 1995 1:49 pm

Lund Software House AB
P.O.Box 7056
S-220 07 Lund, Sweden

Table of Contents

1 Introduction

This library describes a set of classes designed to create a process concept useful when dealing with "soft" real-time problems. This is typically the situations in programs simultaneously handling several input sources like user key-board input, mouse input, a window manager, communication with other programs. These examples are all characterized by the fact that input to the program from the sources may or may not be available at any one time. These situations can sometimes be solved by splitting the program into several communicating UNIX processes. Even so the problem often arises that one such program need to be able to handle more than one input channel. Such a "heavy weight" process solution can also be impractical for other reasons, like the large performance overhead and un-natural communication patterns between heavily dependent processes. There is thus a need for "light weight" processes with fast context switch and possibilities for the processes to share data structures in the same address space. This library is built on the available primitives in Simula and provides an efficient and easy to use process concept understandable in terms of the language.

2 Organisation and use

The routines are distributed as separately compiled Simula classes. The files are normally installed in '/usr/local/simulabin/libsim' (the -.atr files) and the object are reached trhrough the link: '/usr/local/simulabin/lib/ liblibsim.a'.

Declarations in Simula programs:

external class ProcessManager, IOProcess, BasicProcessEvent; ! Always needed ;
external class Monitor, Semaphore; ! Optional;

Compilation:

% simcomp <program> -L=/usr/local/simulabin -I=simioprocess
(or % **simcomp <program> -I**)

Linking:

% simld <programs and other options> -lsimioprocess
(or **%simld <programs and other options> -l**)

3 Overview of functionallity

The process concept presented here is built out of Simula co-routines and each process can wait for input becoming available on one input channel. Input channels are Unix files in general, such as 'pipes' and 'sockets', used for communication between programs, locally or remote. When input is available it will be started in order to do its I/O and further processing. Processes can also specify a maximum time to wait and will receive a special event if no I/O has become available during that period. Processes are communicated with by sending them events. Not all processes have to be tied to an external file, but can as well perform internal tasks. Besides message passing there are also Semaphores and Monitors for implementation of other communication strategies between the processes. Finally there are facilities for generating periodic events, for synchronization with real-time and for orderly shutdown of a system of processes.

The implementation being based on co-routines means that control is explicitly transferred between processes by means of the Simula primitive "Call" and "Detach" statements. These scheduling points are, however, hidden inside a

"process manager" and are not directly visible in the programming interface. This way of implementation results in a very fast process context switch, approximately the same time as a procedure call overhead. The comment above, on "soft" real-time problems, relate to the fact that there is currently no interrupt mechanism in the system. A process given the control will stay in control until it gives it up freely (usually by asking for a new event). This solution is, however, sufficient for a large class of problems. In the future there might be an interrupt mechanism added to the system. This will, however, need changes in the low level implementation of the language, which has so far not been needed. The current system is implemented in pure Simula and thus portable, although the operating system call (to 'select') has to be changed to fit other systems than Unix. The combination of co-routines and the 'select' function enables us to implement a process mechanism without busy-wait. A similar system has been developed for the Macintosh and its window system.

4 System Structure

The library provides two major classes to the user, the IOProcess which is used to create user defined processes by subclassing it and the ProcessManager which performs the scheduling created IOProcess objects. There is also a set of events, in the form of subclasses to the abstract class BasicEvent. Events are sent from the ProcessManager to IOProcesses and between these. Finally there are two classes, Semaphore and Monitor, for communication and synchronization between IOProcess objects.

4.1 ProcessManager

In a single program there will be one and only one object of this class. The task of this object (the PMG) is to schedule IOProcess objects and to hand over control to them when they are ready to execute. It also creates instances of the predefined event classes when appropriate and sends such events to the appropriate IOProcess objects. The PMG maintains a data structure of registered processes and uses a round-robin policy for scheduling if there are more than one IOProcess object ready to execute. If there is no IOProcess to execute, the ProcessManager, and thus the Simula program, will be idle.

4.2 IOProcess

The user will define subclasses of IOProcess where the body will be used to specify the activity of the process. Each such class will thus look like a main program. If it does not communicate with other processes in the same program it can be written with no knowledge of the rest of the processes.

There will typically be a one-to-one correspondence between an interesting file and a process to receive I/O events. This process knows how to initialize the file and how to read from the file. The PMG just notes when something can be read. Other processes might ask for periodic events to work as a clock. It is good practice to keep the processes small handling one external file or one kind of periodic events each.

Note that communication with a window system (like NeWS and X11) is done over a file (pipe, socket,...). In such a situation there will typically be one process (often called the WindowManager) to receive I/O events regarding this file

which will then figure out what to do with it. Typically it will generate some internal event and send it on to a process controlling the corresponding window. This means the WindowManager is just a regular process from the PMG point of view, and the PMG has to take no special action regarding it.

Other interesting processes are the ones that perform reads on pipes from other processes like Servers (and Clients in the other end). In some cases these processes might simply only hand over the control or the read bytes to some other process. This is typically the case when many communication sessions are multiplexed over a pipe between a server and a client.

4.3 Events

Pre defined events are:

- InputEvent – generated when a 'Read' can be performed on a file.
- IOTimeoutEvent – generated if no InputEvent arrived in the specified time interval.
- OutputEvent – generated when a 'Write' can be performed on a file.
- UpdateEvent – generated when the program is about to go idle.
- PeriodicEvent – generated with a requested time interval.
- CancelEvent – generated when the program is to terminate.
- BasicProcessEvent – user defined subclasses being sent from one process to another.

The InputEvent and IOTimeoutEvent events goes together with files used for input from the Simula programs point of view. With these it is possible to write programs that can service an input file with no risk that the program blocks because there is no input available (the input buffer is empty). IOTimeoutEvent events makes it possible to take action if input has not been received for some maximal amount of time (due to communication failure, user in-active etc.).

The OutputEvent and UpdateEvent events goes together with files used for output from the Simula programs point of view. Such programs face the problem that they can be blocked if they generate output to quickly (the bounded output buffer becomes full). The OutputEvent event makes it possible to write event-driven programs also for this situation, keeping the output buffer filled, but not overflowing. In some situations it is attractive for efficiency reasons to use an internal buffer and to write its content to the output file in bigger chunks. The communication with the X11 window system as implemented in Xlib is using this approach. The risk here is naturally that there is a possibility that writing the internal buffer to the file is not done often enough. UpdateEvent events are generated when the PMG is about to become idle and makes it simple to perform such book-keeping activities at the end of a period of processing.

The PeriodicEvent events are generated with a specified interval and can be used to trigger periodic actions, like driving a clock, or in a control situation, reading input values or calculating new control signals. If for some reason the processing is delayed, and a periodic event can not be delivered in time there will be enough events generated to make up for the delay, although they will arrive late no event will be lost.

The pre-defined events described above are all sub-classes of ProcessEvent which in its turn is a subclass of BasicProcessEvents. User-defined events should be defined as sub-classes of BasicProcessEvents. None of the pre-defined events contain any attributes or operations.

4.4 Synchronization

Message-passing with the pre-defined events is used for synchronization between the ProcessManager (and thus the external world) and the IOProcesses. For synchronization among IOProcesses there is the possibility to use user defined events as well as to use the classes Semaphore or Monitor. Semaphore is an implementation of a generalized semaphore based on an integer. Monitor is a slightly more general mechanism than the usual construct. The condition variables in the monitor are implemented by a locally defined class (rather than Boolean variables) with operations 'CauseOne' and 'CauseAll' matching two different semantics of monitor useful in different situations.

5 Examples

Example 1: A clock that generates output every second.
This example shows how a process which does some periodic processing can be implemented.

```
IOProcess class ClockTick;
begin
   EnablePeriodicEvent(1.0); ! seconds;
   while true do
   begin
      inspect WaitEvent
      when PeriodicEvent do
      begin
         Outtext("One scond passed"); Outimage;
      end;
   end;
end --- ClockTick ---;
```

Example 2: Echo of user input with time-out after 60 seconds of inactive user.
This example shows how an input channel (like the one from the user) can be controlled.

```
IOProcess class Echo;
begin
   EnableInputEvents(Sysin);
   while not Sysin.Endfile do
   begin
      StartIOTimeout(Sysin,60); ! Wait max one minute;
      Ev :- WaitEvent;
      inspect Ev
      when InputEvent do
      begin
         Sysin.Inimage;
         Sysout.Outtext("Echo: "); Sysout.Outtext(Sysin.image.strip);
         Sysout.Outimage;
      end
      when IOTimeoutEvent do
      begin
         Sysout.Outtext("Hello, why are you not typing ?"); Sysout.Outimage;
      end - inspect -;
   end - while -;
```

```
end --- Echo --;
```

Example 3: Sending an audio file to the audio device.
This example shows how an output channel can be filled without causing any blocking. It also uses a user defined event.

```
BasicProcessEvent class SoundEvent(SoundName);
   value SoundName; text SoundName;
! ---------------------------------------------------------------- ;
IOProcess class SendSound;
begin
   ref(inbytefile) Sound;
   ref(outbyteFile) Speaker;
   text Bytes;
   Boolean Cancel;
      Speaker:-new outbytefile("/dev/audio");
   Bytes:-blanks(1024);
   if not Speaker.open then
   begin
      outtext("SendSound: can't open '/dev/audio'"); outimage;
      Terminate;
   end;
   EnableCancelEvents;
   while not Ev in CancelEvent do
   begin
      Ev :- WaitEvent;
      inspect WaitEvent
      when SoundEvent do
      begin
         Sound :- new inbytefile(SoundName);
         if Sound.open then
         begin
            EnableOutputEvents(Speaker);
            while not Sound.endfile and not Ev in CancelEvent do
            begin
               Ev :- WaitEvent;
               inspect Ev
               when OutputEvent do
                  Speaker.Outtext(Sound.Intext(Bytes) );
            end - while -;
            DisableOutputEvents(Speaker);
            Sound.Close;
         end;
      end;
   end -- while --;
   Speaker.Close;
end -- SendSound --;
```

Example 4: Producer, consumer example communicating over a buffer.
This example shows how processes can synchronize using a subclass of Monitor. It is also an complete program, showing how initial processes are created and scheduling is started.

```
begin
   external class ProcessManager, IOProcess, Monitor;
! ---------------------------------------------------------------- ;
```

```
    Monitor class Buffer(Size); integer Size
    begin
       ref(Condition) NonFull, NonEmpty;
       integer array Buffer(0:Size-1);
       integer inP, outP;
       procedure Put(x); integer x; ! ----------------------;
       begin
          EnterMonitor;
          while not mod(InP+1,Size)=OutP do
             NonFull.Await;
          Buffer(InP):=X;
          InP:=mod(InP+1,Size);
          NonEmpty.CauseOne;
          ExitMonitor;
       end -- Put --;
       integer procedure Get;  ! ----------------------;
       begin
          EnterMonitor;
          while not InP=OutP do
             NonFull.Await;
          Get:=Buffer(OutP);
          OutP:=mod(OutP+1,Size);
          NonFull.CauseOne;
          ExitMonitor;
       end -- Put --;
       NonFull:- new Condition;
       NonEmpty:-new Condition;
       InP := OutP := 0;
    end --- Buffer ---;
! ------------------------------------------------------------------- ;
    IOProcess class Producer(Buff); ref(Buffer) Buff;
    begin
       integer U,X;
       U:=1147;
       while true do
       begin
          X:= Randint(U,1,100);
          B.Put(X);
       end;
    end --- Producer ---;
! ------------------------------------------------------------------- ;
    IOProcess class Consumer(Buff); ref(Buffer) Buff;
    begin
       integer x;
       while true do
       begin
          X:=Buff.Get;
          outint(X,0); Outimage;
       end;
    end --- Consumer ---;
! ------------------------------------------------------------------- ;
! Startup:;
    ref(ProcessManager) PMG;
    ref(IOProcess) P,C;
```

```
    ref(Buffer) B;
    ! This part is executed sequentially: ;
    PMG :- new ProcessManager;
    B:- new Buffer(PMG);
    P:- new Producer(B);
    C:- new Consumer(B);
    PMG.RegisterProcess(P);
    PMG.RegisterProcess(C);
    P.Start;
    C.Start;
    ! At the 'Run' the control is handed to the PMG and process scheduling
starts;
    PMG.Run; ! The 'main program' waits at this point;
    ! When the scheduling is terminated the sequential processing is
    ! returned to this point;
    Outtext("Good bye");
    Outimage;
end
```

6 Abstract class descriptions

6.1 class ProcessManager

Terminology

Registered processes – These are the IOProcesses objects that has been registered with the manager. These are the processes that will take part in the scheduling. Typically IOProcesses are registered at creation and de-registered at their termination.

ReadyQ – This queue is keeping track of all processes that currently are ready to execute.

Current Process – The one IOProcess object in the ReadyQ that is currently executing,e.i. in the 'Running' state. This object will continue to execute until it executes one of the operations and leaves the Ready/Running state as shown in the figure below. It might also enter the Running state by executing a Pause operation and leaving control to some other process in the ReadyQ.

Known Files – When the ReadyQ is empty the manager is waiting for input arriving to any of these files. When so happens the corresponding IOProcess is notified with an InputEvent. The manager maintains the association between the file and the IOProcess object. KnownFiles can be either Simula files or plain Unix files.

IOTimers – For each KnownFile there might also be specified a certain time before which input should occur. If not, the manager notifies the corresponding IOProcess with an IOTimeoutEvent. A IOProcess that has asked for IOTimeOutEvent will receive either an InputEvent during the interval or an IOTimeoutEvent at the end of the interval (but never both). A started timer is cancelled by either of these events and has to be restarted to be effective again. The IOTimer is not effected should other events arrive.

Update events – when the manager is about to become idle (any other events has been generated and no process is ready to be executed) the manager will generate UpdateEvents to all the processes which have enabled such events.

Periodic events – An IOProcess can also ask for events with a certain time interval. The manager will generate PeriodicEvents and send them to the process

accordingly. The manager is maintaining a set of all timers and periodic events. The closest in time of these specifies the longest time the manager can stay idle when the ReadyQ is empty.

Cancel Events – CancelEvents are be broadcasted to all IOProcesses that have registered for such events. The broadcasting is triggered by a call to the manager. IOProcesses should respond to such events by orderly shut down any activities and Terminate or Stop. All IOProcesses that have registered files of periodic timers should also register for CancelEvents in order to perform an orderly shutdown of the system.

6.2 class IOProcess.

Terminology

Execution state – The IOProcesses can be in any of five states. The transition between these states are shown in the figure below. The meaning of the states are as follows:

- *Running* - The one and only process that is actually executing at a given time.
- *Ready* - All the processes that are "ready" to execute. Currently there is no pre-emption so the CurrentProcess is allowed to execute until it freely gives up the initiative. In the future the system might include an interrupt mechanism to automatically distribute the execution time over all the ready processes and thus move processes between the Running and Ready state.
- *Suspended* - This is an idle state where the process is waiting to be started by another process. This mechanism is used in the implementation of Semaphores and Monitors.
- *Waiting* - This is an idle state where the process is waiting for an Event to arrive. An Event can be sent either by another process or by the manager.
- *Terminated* - A process in this state can not be moved to any other state again. It can enter this state from any of the other states. When being terminated the process is automatically de-registered from the manager and all timers etc. are cancelled.

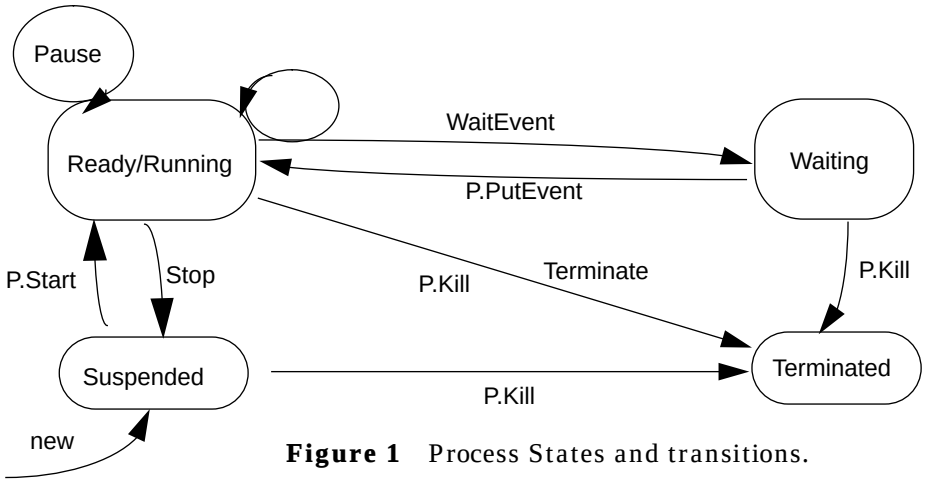


Figure 1 Process States and transitions.

Note the notational difference: P.Kill indicates that the operation is performed by another process than the effected one, while Terminate means that a process is initiating this operation when it is itself is in the runnng state.

Event Queue – Each process maintains a queue of events to process. When this queue is empty a call to WaitEvent transfers the process from Ready to the Waiting state.

Priority – Each IOProcess object has an associated priority. This priority can be used to give execution time more frequently or in longer periods to some processes. Priority can be used to have execution time demanding processes to coexist with processes requiring fast response. Better priority is not a guarantee to get execution first and should not be used as a substitute for proper synchronization. Processes are created with priority 0 which is "best". The priority should be set to a higher value for execution demanding processes

6.3 class Semaphore

Terminology

Resources – Each Semaphore uses an integer to represents a finite resource. Semaphores with single resources can be used to ensure mutual exclusion between processes.

ProcessQueue – Processes asking for unavailable resources are delayed until the resources are returned to the Semaphore. These processes are suspended and kept in a queue until they can be given the resource and started again.

6.4 class Monitor

Terminology

Mutual Exclusion – Only one process may have access to the operations of the Monitor at any given time. Processes trying to call a Monitor operation when it is already occupied are delayed until the Monitor is free again. Delayed processes are then let into the Monitor one at the time.

EnterQueue – This is a queue of processes waiting to enter the monitor.

Condition – A Condition is representing a certain state of the data contained in the Monitor. A process inside a Monitor can choose to wait for such a state to occur. As they do so they are leaving the Monitor but kept track of by the Condition. When the desired state is reached this is signalled to the Condition and the waiting processes are transferred to the EnterQueue to enter the Monitor when it becomes free.

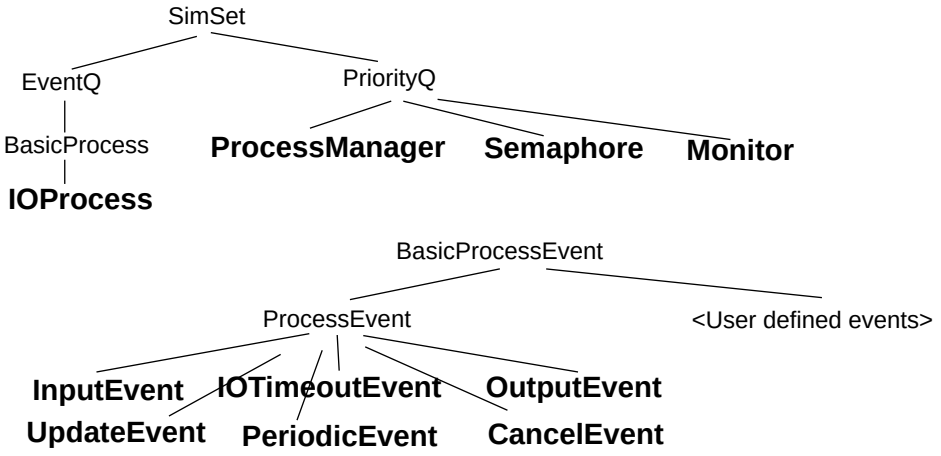


Figure 2 *simlib ioprocesses class hierarchy*

7 Class hierarchy.

In fig. 2 the hierarchy of classes used in this library are shown. Names in thin font indicates abstract classes, invented for technical purposses only. They are not descibed in detail.

8 Detailed Interfaces

8.1 Process Manager

PriorityQ class ProcessManager;

IOProcess objects must be 'registered' in order to be managed by this scheduler. When processes ready to run are avilable they are activated. When there are no such processes this schedluer makes the Simula program sleep until someting happens (input avilable or timer goes off). When there are no open files watched by a IOProcess and there are no timers set, the scheduling finishes. CancelAll all is a 'soft' terminater allowing those who need to cleanup to do so. Terminate-All is a killing all IOPorcesses directly.
Supers: PriorityQ
Kind: SingelObject
Init: At least one IOProcess must be started before 'Run' is called.
'Run' works as a scheduler and 'Call's other IOProcess based on their state.
Sequencing: (RegisterProcess
: (Run / RunningProcess / CancelAll / TerminateAll)*
: DeRegisterProcess)*

User accessable operations

RegisterProcess

procedure RegisterProcess(P);
ref(BasicProcess) P;

Register P as a process that can be scheduled. P is an object of a subclass of IOProcess (see below).

DeRegisterProcess

procedure DeRegisterProcess(P);
ref(BasicProcess) p;

Forget about P - It can not be scheduled again unless re-registered. Terminated IOProcess objects are automatically de-registered.

Run

procedure Run;

Give control to this ProcessManager -- it will schedule processes and call them as appropriate. Run will finish when 'nothing can ever happen again' or after a call to TerminateAll. When a call to Run is finished the execution will continue after the call to Run.

RunningProcess

ref(basicProcess) procedure RunningProcess;

return a reference to the currently executing process. (Mainly intended for use in implementation of scheduling primitives such as Semaphore and Monitor. Should not be neccessary to use in the applications).

CancelAll

procedure CancelAll;

Broadcast CancelEvent:s to all IOProcesses that have called EnableCancelE-vent. Besides this the PMG does not do anything exceptional. Processes with EnableIO- Events should as a result call DisableIOEvents when terminating actions are done. As a result of broadcasted CancelEvent:s there should after a while be no processes left in the Ready state, with IO-Events enabled or with timers set, so the terminating condition of the Run-procedure should be fullfiled. Execution is then resumed after the call to Run in 'the main program'. Cancel-All can be called many times if the processes refuses to die... If the processes are left in a consistent state a new call to Run should be possible.

TerminateAll
procedure TerminateAll;

Emergency termination of the Run-call. Control is immediately transferred to the point after the call to Run in 'the main program'. There are also a number of operations in the ProcessManager that are called from class Process and should not be called by the user code directly.

8.2 Events

The Events are defined without attributes. It should always be clear from the situation what the event indicates. This design choice is consistent with the idea that IOProcess should be rather small and do one well-defined task. It is thus not possible to have a IOPorcess that waits for events from two different external files, but one can have two IOProcesses waiting for an external file each!.

BasicProcessEvent
class BasicProcessEvent;

Abstract superclass of all Events, used as superclass for user defined events.

ProcessEvent
BasicProcessEvent class ProcessEvent;

Abstract superclass of all Event classes generated by the Process Manager.

InputEvent
ProcessEvent class InputEvent;

Generated when an inimage (or read) can safely be issued on the file for which EnableIOEvents has been called.

OutputEvent
ProcessEvent class OutputEvent;

Generated when the output buffer is non-full and a write can be issued on the file for which EnableIOEvents has been called.

IOTimeoutEvent
ProcessEvent class IOTimeoutEvent;

Generated when the timeout period has passed with no InputEvent generated. StartIOTimeout must have been called. A process will receive a IOTimeoutEvent or a InputEvent, but never both

UpdateEvent
ProcessEvent class UpdateEvent;

Generated when some other IOProcess have recieved some Event and this process did not become active during its processing

PeriodicEvent
ProcessEvent class PeriodicEvent;

Generated when the propper time interval has passed.

CancelEvent
ProcessEvent class CancelEvent;

Broadcasted to all IOProcess object that have requested them with a call to EnableCancelEvents. Events are triggered by a process calling the ProcessManager operation CancelAll.

8.3 IOProcess

BasicProcess class IOProcess;

There will typically be a one-to-one correspondence between an interesting file and a process to receive I/O events. This process knows how to initialize the file and how to read from the file. Other processes might ask for periodic events to

work as a clock. It is good practice to keep the processes small handling one external file or one kind of periodic events each.
An IOProcess object is created by another IOProcess (or the main program), initialized and registered by the ProcessManager, and the Started.
It can be Stop either by itself (calling Terminate or pass out through its final end), or it can be stopped by another process (calling its 'Stop').
Of the many events, there are typically just a few relevenat for a single class of IOProcesses.
Supers: BasicProcess, EventQ, Simset
Kind: Abstract
Init: The body of the subclass will be executed as a co-routine/process
Sequencing: new <sync init> PMG.RegisterProcess(this IOProcess) Start
: (A* Stop Start)* Kill/Terminate/end-of-class
: A= Pause / Nice / WaitEvent / AskEvent / PutEvent /
: EnablePeriodicEvents / DisablePeriodicEvents /
: EnableCancelEvents / DisableCancelEvents /
: EnableUpdateEvents / DisableUpdateEvents /
: EnableInputEvents / StartIOTimeout / DisableInputEvents /
: EnableOutputEvents / DisableOutputEvents /
: EnableInputEventsUNIX / StartIOTimeoutUNIX / DisableInputEventsUNIX /
: EnableOutputEventsUNIX/ DisableOutputEventsUNIX /

External operations only

Start
procedure Start;

Put the process into the Read/Running state. The process will be given chance to execute in due time according to priorities and scheduling policies. This operations is intend to be invoced from outside the IOProcess object only, typically by the creator, after some initializations (a process can not start itself).

Kill
procedure Kill;

This operation causes the process to become terminated. It is thus analogous to the operation Terminate but Kill can be called from another process and the killed process can be in any state of execution.

Internal (protected) operations

Stop
procedure Stop;

Move the process into the Suspended state. This means that the process will nou execute until it has been Start:ed again. Stop can be called (direct or indirect) only when the process itself is executing.

Nice
procedure Nice(N); integer N;

Lower the priority of this IOProcess. Initially processes have the same, high, priority. Nice is called by background processes not to disrupt response-critical processes to much. They should have Pause-calles sprinkled over time consuming parts. Notice that Nice can NOT be used for solving synchronization problems. There is no guarantee that a nicer process is not executed when other processes are ready, it is just a bit less likely.

Terminate
procedure Terminate;

Stop execution of this IOProcess and deregister it.

WaitEvent
ref(BasicProcessEvent) procedure WaitEvent;

Wait for next event to arrive, then return with the Event. If no event is queued the IOProcess is blocked until an Event has been sent to this process with a call to PutEvent.

AskEvent
Boolean procedure AskEvent;
This function returns True if any event pending, False otherwise.

PutEvent
procedure PutEvent(E);
ref(BasicProcessEvent) E;
Queue the ProcessEvent, E, for this process. This will trigger the IOProcess to be moved to the Ready/Running state if it has called the operation WaitEvent and no event was pending.

Pause
procedure Pause;
Simulation of interrupt used to trigger rescheduling. Insert this into code where the execution tends to take a lot of time. Note: Beware of concurrency problems and ensure mutual exclusion!

Event control

EnableCancelEvents
procedure EnableCancelEvents;
Enable CancelEvent to be received as broadcasted by a call to CancelAll of the PMG.

DisableCancelEvents
procedure DisableCancelEvents;
Called to not get any future CancelEvents.

EnablePeriodicEvents
procedure EnablePeriodicEvents(Seconds); long real Seconds;
Order PeriodicEvents to appear with a distance of Sec seconds on the average starting Sec seconds from now. Earlier enables are replaced by a new request. Note periods longer than 24 hours are not allowed. If for some reason the event can not be generated at the propper time one (or several) will be generated as soon as possible to make up for the passing time.

DisablePeriodicEvents
procedure DisablePeriodicEvents;
Cancels any ordered PeriodicEvent.

EnableUpdateEvents
procedure EnableUpdateEvents;
Order UpdateEvent. They are generated when the Scheduler has nothing more to do and is about to go idle. Can be used to trigged clean up activities such as emptying output buffers etc before the system goes to sleep, wating for some external (Periodic, Input, IOTimeout or Output -) Event.

DisableUpdateEvents
procedure DisableUpdateEvents;
Cancels generating UpdateEvents to this IOProcess.

EnableInputEvents
procedure EnableInputEvents(F);
ref(File) F; ! Simula In(byte)file or Direct(byte)file;
Enable events related to the Simula (input or directaccess) File F. F must have been opened before the call. The IOProcess will receive an InputEvent when an 'Inimage' (or Intext) can be performed.

StartIOTimeout
procedure StartIOTimeout(F,Sec);
ref(File) F; ! Simula In(byte)file or Direct(byte)file;
Real Sec; ! Maximum time to wait for next input.;

Start a timer on the file F with duration Sec Seconds. (F must previously been enabled with a call to EnableInputEvents). The effect of the timer is that an IOTimeoutEvent will appear after the time Sec if not an InputEvent has been generated before. One and only one of these two possible events will appear. The timer is started at the time of the call of this operation and it have only effect once.

DisableInputEvents
procedure DisableInputEvents(F);
ref(file) F; ! Simula In(byte)file or Direct(byte)file;

Forget about the Simula file F. This operation is typically called before the file is closed.

EnableOutputEvents
procedure EnableOutputEvents(F);
ref(File) F; ! Simula Out(byte)file or Direct(byte)file;

Enable events related to the Simula (output or directaccess) File F. F must have been opened before the call. The IOProcess will receive an OutputEvent when an 'Outimage' (or 'Outtext') can be performed.

DisableOutputEvents
procedure DisableOutputEvents(F);
ref(file) F; ! Simula Out(byte)file or Direct(byte)file;

Forget about the Simula file F. This operation is typically called before the file is closed.

UNIX related operations

These Five operations have the same effect as the corresponding operations above but relate to standard UNIX files. N is a Unix file system number as returned by open.

EnableInputEventsUNIX
procedure EnableInputEventsUNIX(N);
integer N; ! Unix file number (0,1,2..) as returned by open.;

StartIOTimeoutUNIX
procedure StartIOTimeoutUNIX(N,Sec);
integer N; ! Unix file number (0,1,2..) as returned by open.;
Real Sec; ! Maximum time to wit for next input operation. ;

DisableInputEventsUNIX
procedure DisableInputEventsUNIX(N);
integer N; ! Unix file number (0,1,2..) as returned by open.;

EnableOutputEventsUNIX
procedure EnableOutputEventsUNIX(N);
integer N; ! Unix file number (0,1,2..) as returned by open.;

DisableOutputEventsUNIX
procedure DisableOutputEventsUNIX(N);
integer N; ! Unix file number (0,1,2..) as returned by open.;

8.4 Semaphore

PriorityQ class Semaphore(PMG);
ref(ProcessManager) PMG;

This class is used for traditional Dijkstra Semaphore synchronization. Supply reference to the ProcessManager as parameter.
Supers: PriorityQ, SortedPool, Simset
Kind: Instantable
Init: PMG must reference the SingleObject ProcessManager.
Sequencing: (Signal / Wait)*

Operations

Signal
procedure Signal;

Normal meaning for Semaphores - Start one waiting process (if any), increase resources.

Wait
procedure Wait;

Normal meaning for Semaphores - Suspend caller if resources=0, decrease resources.

8.5 Monitor

PriorityQ class Monitor(PMG);
ref(ProcessManager) PMG;

This class is an implementation of (almost) traditional monitors. Monitors are used to synchronize processes operating on shared data by means of the Monitor operations. Only one IOProcess is allowed to execute a Mointor operation at the time. Supply a reference to the ProcessManager as parameter.
Supers: PriorityQ, SortedPool, Simset
Kind: Instantable
Init: PMG must reference the SingleObject ProcessManager.
Sequencing in Mointor operations (where c is a Condition object):
: EnterMonitor
: (c.Await/c.CauseOne/c.CauseAll)*
: ExitMonitor

Requiered operations to call from accessable procedures

EnterMonitor
procedure EnterMonitor;

Must be called at the entry of each monitor operation.

ExitMonitor
procedure ExitMonitor;

Must be called at the exit of each monitor operation.

Local nested class

Condition
class Condition;

This class is local to the Monitor. Create object of this class to represent conditions (to wait for) in the monitor.
Sequencing: (Await / CauseOne / CauseAll)*;

Operations on Conditions

Await
procedure Await;

Await the conditon to be fullfiled.

CauseOne
procedure CauseOne;

The condition is true, start (at most) one waiting process.

CauseAll
procedure CauseAll;

The condition is true for all waiting processes - Start them all.

9 Index to classes and procedures