CS 452/652

Revised Spring 1989

Project 1 Specification

The following is a complete specification of a real-time operating system which supports concurrent programming by message-passing and multitasking. It consists of four parts:

1. Domain types
2. Kernel primitives
3. System primitives
4. Input/Output and Clock primitives

The primitives specified in (2) are implemented inside the kernel; (3) are implemented as system processes created by the operating system; (4) are implemented either inside the kernel or as system processes. Based on real time constraints, you have to decide how these primitives are implemented.

Primitives that are *required* to be implemented are specified in **bold** face. Optional (i.e. not required) primitives have a superscript †. We sometimes use the words “process” and “pid” (process id) interchangeably.

0.1 Domain Types

To specify the functionality of each primitive, we need the following domain types.

Name = a set of valid identifiers;

Priority = {0,1,2,...,RANK}, where RANK is the highest and 0 is the lowest process priority;

Pid = {1,2,...,MAX}, where MAX is the largest process id that will fit into 16 bits;

Event = a set of possible events (or interrupts), including {keyboard, display, timer,...};

Directory = a name directory including {train-controller, display-server,...};

Byte = a set of unsigned 8-bit quantities;

Message = a set of finite sequences of *Byte*;

Length = {0,1,...,MAXLEN}, where MAXLEN is the maximum length;

Port = a set of input/output ports, including {serial-in, serial-out,...};

Tick = {0,1,2,...}, where a unit of time depends on the resolution of the timer;

Bool = {true, false};

Void = {null}.

To capture the possibility of errors, there is a special element called *bottom*, denoted by $\perp$, which specifies an error.

0.2 Kernel Primitives

The kernel is the nucleus of a real-time operating system. Its main functions include CPU allocation, context switching, process creation and destruction, process scheduling, message routing, event handling, resource allocation, and possibly device handling. These functions can be achieved either exclusively by the kernel or by some system processes.

CPU allocation Processes are allocated the CPU as follows.

1. A process eligible for execution is said to be *ready*.
2. A process which is not ready is said to be *blocked*.
3. Every process has a fixed process priority. Among the highest priority ready processes, the process ready for the longest time is allocated the CPU, and is said to be *active*.
4. The active process relinquishes the CPU either by becoming blocked (see 5), by explicitly yielding the CPU, by implicitly yielding the CPU through round robin scheduling of equal priority processes (optional), or by being pre-empted when a higher priority process becomes ready. The latter is either caused by a hardware interrupt or by an action of the active process.
5. The active process is blocked either by attempting to communicate with another process or by waiting for an event (or interrupt) to occur.

Process creation A process is created with a specified name and priority.

$\boxed{\textbf{Create} : (\textit{Name}\cup\{\bot\})\times(\textit{Priority}\cup\{\bot\})\rightarrow(\textit{Pid}\cup\{\bot\})}$

Create($\bot$,p) = $\bot$;

Create(n,$\bot$) = $\bot$;

Create(n,p) = pid, where *pid* is the process id of a newly-created process associated with the procedure named n; initially, the state of *pid* is *ready*, its priority is p; process just created is a descendant of its creator—the current active process;

Create(n,p) = $\bot$, otherwise.

Process destruction Any process can destroy any other process as long as it knows the other process' pid. A process is allowed to destroy itself.

$\boxed{\text{Destroy}^{\dagger} : (\textit{Pid}\cup\{\bot\})\rightarrow(\textit{Void}\cup\{\bot\})}$

Destroy($\bot$) = $\bot$, *no action taken*;

Destroy(pid)= *null*, the process *pid* and all its descendants are removed permanently from the system; the resources and data storage occupied by these processes are reclaimed; all other processes that are blocked on these processes become ready;

Destroy(pid)= $\bot$, otherwise.

Process termination If a process finishes its work, it should inform the kernel so that it gets no more CPU time.

$\boxed{\textbf{Exit} : \{\}\ \textit{(DOES NOT RETURN)}}$

Exit() No semantics are implied other than that the calling process will never again be active.

CPU yielding If there are several processes at the same priority, and round robin scheduling is not done, system performance may be improved by having a process yield the CPU to the process with the same priority that has been ready the longest.

$\boxed{\textbf{Pass} : \{\}\rightarrow\textit{Void}}$

Pass() = *null*, the calling process remains ready and the process with the same priority (as the calling process) that has been ready the longest is made active.

Process identification Six primitives are defined for finding or checking the pid and priority of a process. Four are kernel primitives and the other two are system primitives. The following are the kernel primitives. We shall specify the other two in a later section.

$\boxed{\textbf{ValidPid} : (Pid \cup \{\perp\}) \rightarrow (Bool \cup \{\perp\})}$

ValidPid($\perp$) = $\perp$;

ValidPid(pid) = *true*, if *pid* is one of the existing processes;

ValidPid(pid) = *false*, otherwise.

$\boxed{\textbf{MyPid} : \{\} \rightarrow Pid}$

MyPid() = pid, where *pid* is the process id of the requesting process.

$\boxed{\textbf{MyParentPid} : \{\} \rightarrow Pid}$

MyParentPid() = pid, where *pid* is the process id of the parent (creator) of requesting process.

$\boxed{\textbf{MyPriority} : \{\} \rightarrow Priority}$

MyPriority() = p, where p is the process priority of the requesting process.

Interprocess communication There are three primitives for passing messages between processes. A process sends a message to another by

$\boxed{\textbf{Send} : (Pid \cup \{\perp\}) \times Message \times Length \times Message \times Length \rightarrow (Void \cup \{\perp\})}$

Send($\perp$,msg,m,rpy,r) = $\perp$;

Send(pid,msg,m,rpy,r) = null, the content of the message msg with length m is sent to the process *pid*; the sending process is blocked until *pid* has received the message and provided a reply into *rpy* by **Reply**;

Send(pid,msg,m,rpy,r) = $\perp$, if the reply message length is $> r$;

Send(pid,msg,m,rpy,r) = $\perp$, otherwise.

Note that msg and *rpy* can share the same message buffer. In such a case, the reply message overwrites the original message. The receiving process can receive a message from *any* other process by

$\boxed{\textbf{Receive} : Message \times Length \rightarrow (Void \cup \{\perp\}) \times Pid}$

Receive(msg,m) = (null, pid), the receiving process may become blocked to receive a message of length $\leq m$ in msg; *pid* is the process whose message msg arrives first, i.e., *first-come-first-serve.*

Receive(msg,m) = ($\perp$, pid), if the length of the sent message from process *pid* is $> m$;

To receive a message from a specific process, the receiving process can specify its pid as in

$\boxed{\text{ReceiveSpecific}^{\dagger} : (Pid \cup \{\perp\}) \times Message \times Length \rightarrow (Void \cup \{\perp\})}$

ReceiveSpecific($\perp$,msg,m) = $\perp$;

ReceiveSpecific(pid,msg,m) = *null*, the receiving process may become blocked to receive a message in msg of length $\leq m$ from *pid* specifically;

ReceiveSpecific(pid,msg,m) $= \perp$, otherwise.

In order to unblock the sending process, the receiving process *must* reply with a message by

$\boxed{\textbf{Reply} : (Pid \cup \{\perp\}) \times Message \times Length \rightarrow (Void \cup \{\perp\})}$

Reply($\perp$,rpy,r) $= \perp$, *no action taken*;

Reply(pid,rpy,r) $=$ *null*, the reply message *rpy* of length r is sent to the specified process *pid* awaiting the reply from the receiving process; the process *pid* becomes ready upon receiving the reply; the replying process is *never* blocked;

Reply(pid,rpy,r) $= \perp$, otherwise.

Interrupt handling All external interrupts are abstracted as events. A process waits for an occurrence of an interrupt by calling AwaitEvent() with the correct event number. AwaitEvent() returns

$\boxed{\textbf{AwaitEvent} : (Event \cup \{\perp\}) \rightarrow (Void \cup \{\perp\} \cup Byte)}$

AwaitEvent($\perp$) $= \perp$, *no action taken*;

AwaitEvent(event) $=$ *null*, for events with no associated volatile information.

AwaitEvent(event) $=$ *byte*, for events with associated volatile information.

0.3 System Primitives

The following set of primitives is used for process identification purpose. It provides a simple form of name service. A process can find out another process' pid if it knows the process' name. A process can make its pid available to others by registering its name in the name directory—it is very much like a telephone directory. To register its name, a process makes a request of

$\boxed{\textbf{RegisterAs} : (Directory \cup \{\perp\}) \rightarrow (Void \cup \{\perp\})}$

RegisterAs($\perp$) $= \perp$;

RegisterAs(n) $=$ *null*, the caller's *pid* is entered into the name table as n; the mapping from names to pids is a multi valued function. In otherwords, a single *pid* may map into more than one *name*, but each name maps to a single *pid*.

RegisterAs(n) $= \perp$, if n is already mapped.

To find out the pid of a process in the name directory, a process makes a request of

$\boxed{\textbf{WhoIs} : (Directory \cup \{\perp\}) \rightarrow (Pid \cup \{\perp\})}$

WhoIs($\perp$) $= \perp$;

WhoIs(n) $=$ pid, where *pid* is the process id with the registered name n;

WhoIs(n) $= \perp$, otherwise.

0.4 Input/Output and Clock Primitives

There is a set of input/output devices, such as keyboard, screen, etc. To input a byte from an input port, a process makes a request of

$\boxed{\textbf{Get} : (Port \cup \{\perp\}) \rightarrow (Void \cup \{\perp\}) \times Byte}$

Get($\perp$) = ($\perp$, b), *no action taken*;

Get(p) = (null,b), where b is the next byte input from input port p; the requesting process is blocked until b is available;

Get(p) = ($\perp$,b), otherwise.

Note that the end of input from an input port is left unspecified. To output a byte to an output port, a process makes a request of

$\boxed{\textbf{Put} : (Port \cup \{\perp\}) \times Byte \rightarrow (Void \cup \{\perp\})}$

Put($\perp$,b) = $\perp$, *no action taken*;

Put(p,b) = *null*, where b is the next byte output to output port p; the requesting process is blocked until b is output or buffered;

Put(p,b) = $\perp$, otherwise.

There is also a general timer device for timing purpose. To idle for a short period of time, a process makes a request of

$\boxed{\textbf{Delay} : Tick \rightarrow (Void \cup \{\perp\})}$

Delay($\perp$) = $\perp$, *no action taken*;

Delay(0) = *null*, *no action taken*;

Delay(t) = *null*, the requesting process blocks until t ticks have been decremented to 0.

In addition to the above primitives the Standard Kernel also implements three additional primitives for the SerialServer. These are: **Read, Write,** and **Config.** Read and Write are used to read/write blocks of data to/from the SerialServer. Config is used to configure the SerialServer for various timeout modes and to use the Robot port instead of the Train port. See their related man pages for details of their implementation and interface.