Project Specification for Computer Science 314

Jacques Eloff (eloff@cs.sun.ac.za)
Department of Computer Science
University of Stellenbosch

March 2004

Contents

1 Introduction

A small, dedicated operating system for the Intel 80386 and compatible platforms must be developed. The system must provide a number of features including:

- the concurrent execution of user-level processes.
- the provision of protected address spaces for user-level processes.
- mechanisms to support synchronous process communication.
- a minimum set of system calls to facilitate I/O operations by user-level processes.

2 Design Constraints

The system must be designed to operate correctly and efficiently on any system meeting the minimum hardware criteria and must be designed for optimal execution and minimal memory consumption. The target platform on which the software will be executed must meet the following minimum requirements:

CPU The system will contain an Intel 80386 or compatible processor.

Memory The system will contain at least 1MB of memory.

Storage The system will contain a $3\frac{1}{2}$" 1.44MB diskette drive along with a NEC 765, Intel 8272A or compatible diskette controller.

Input/Output The system will contain either a monochrome (MDA) or colour (VGA) display adapter as its primary output source and will use a standard 101-key keyboard as its primary input source.

3 System Structure

The system will be composed from a number of modules as listed in Table 1. Some of these modules have already been provided (**P**), while others may require additional extensions (**E**) or must still be implemented (**I**). The module dependencies are illustrated in Figure 1 using a directed graph. Modules are represented by vertices and the import dependencies as directed edges. A module w is dependent on (imports) a module v if there is a directed edge (v, w) between the two modules. Note that `osTrace` has been omitted because it is primarily used for debug output.
A number of user-level libraries must also be implemented and will be used when implementing user-level processes. These libraries include `TTY`, `IPC` and `SYS`. The `TTY` library forms the user-level interface to communicate with the I/O sub-system implemented by `osTTY` while `IPC` forms the user-level interface to access the communications primitives supported by `osComms` to facilitate synchronous process communication. The `SYS` library contains general system related functions.

4 Functional Requirements

4.1 Module: osStorage

This module must provide the necessary mechanisms to support various memory management structures. Its primary function is to allocate and deallocate blocks of memory that can be used for dynamic structures inside the kernel such as process queues. It must also provide mechanisms to manage pages and DMA buffers.

```
PROCEDURE Available*(): LONGINT;
```

Module	Description	Status
osBootTable	Provides access to boot table information	P
osTrace	Basic display primitives for debugging	I
osMachine	Abstraction of hardware	P
osStorage	Memory management	I
osProcess	Process management	I
os8259	Interrupt management	P, E
osExceptions	Exception handling	P
osDMA	Driver: DMA Controller	I
osTTY	I/O sub-system	P
osTimer	Driver: System clock	I
osFloppy	Driver: Floppy diskette	P
osLoader	Loads user-processes from disk	I
osComms	Synchronous process communication	I
osSysCalls	Dispatches system calls to appropriate modules	P, E
osInit	System initialization	P, E

Table 1: System modules and their function.

Description This function will return the total amount of memory available for allocation and will be reported in bytes. The reported amount includes memory available for DMA buffers, pages and dynamic structures.

```
PROCEDURE LargestAvailable*(): LONGINT;
```

Description This function will return the size of the largest contiguous block of memory that is available for allocation. The size of this block will be reported in bytes.

```
PROCEDURE AllocateDMA*(size, channel: LONGINT): LONGINT;
```

Description This function will be called by device drivers to allocate memory for a DMA buffer. The function must adhere to the following behaviour:

- The function will return the address of the buffer that was allocated if the operation was successful. The function will guarantee that the buffer will be allocated in a valid memory region given the DMA channel associated with the buffer.
- The function will return 0 if the DMA channel is invalid.
- The function will return 0 if the required size exceeds the buffer capacity for the specific channel or if the required size is less than or equal to 0.
- The function will return 0 if there is not enough memory to satisfy the request.

```
PROCEDURE NewRec(VAR p: ADDRESS; tag: Tag);
```


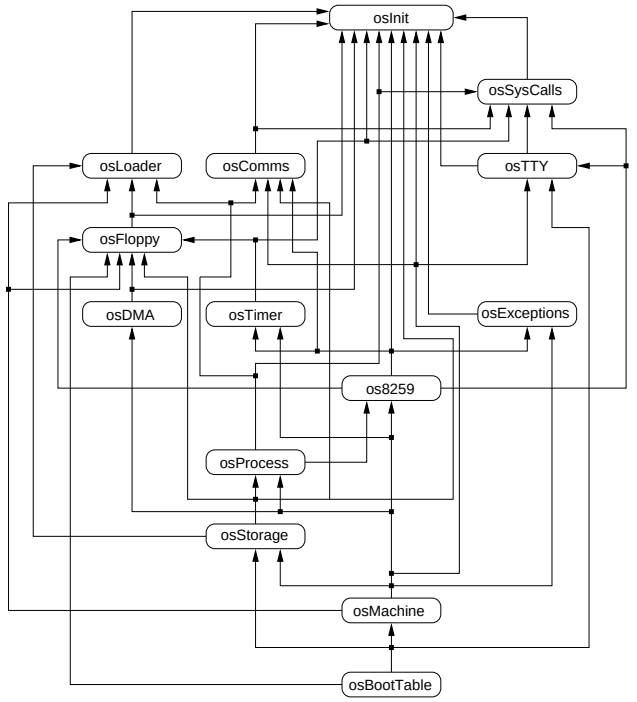

Figure 1: Module dependencies.

Description All calls to the built-in `NEW` procedure will be patched by the Oberon linker to reference this procedure if the parameter associated with `NEW` is a pointer to a `RECORD` structure. The `tag` parameter is automatically generated by the compiler and will contain information about the dynamic structure. The procedure must adhere to the following behaviour:

- The procedure will allocate a suitable block of memory from the pool of available memory blocks and assign the start address of the allocated block to `p`.
- The procedure will minimize the amount of internal and external fragmentation that may occur as a result of the memory allocation process.
- The function will set `p` to 0 if a suitable block could not be located.
- The function will set `p` to 0 if the block size specified in `tag` is less than or equal to 0.

```
PROCEDURE NewSys(VAR p: ADDRESS; size: LONGINT);
```

Description All calls to the built-in NEW procedure will be patched by the Oberon linker to reference this procedure if the parameter associated with NEW is a pointer to an open array. The size parameter is automatically generated by the compiler and includes the necessary space for the type descriptor associated with the open array. The procedure must adhere to the following behaviour:

- The procedure will allocate a suitable block of memory from the pool of available memory blocks and assign the start address of the allocated block to p.
- The procedure will minimize the amount of internal and external fragmentation that may occur as a result of the memory allocation process.
- The function will set p to 0 if a suitable block could not be located.
- The function will set p to 0 if the block size specified in size is less than or equal to 0.

```
PROCEDURE AllocatePage*(VAR address: ADDRESS);
```

Description This procedure will be called to allocate fixed size memory pages for page directories, page tables and processes. The function must adhere to the following behaviour:

- The procedure will set address to the start address of the page that was allocated if the operation was successful. The address will correspond to a physical address.
- The start address of the page will be aligned on a 4K boundary and will occupy exactly 4096 bytes.
- The procedure will set address to 0 if a page could not be successfully allocated.

```
PROCEDURE FreePage*(address: ADDRESS);
```

Description This procedure will be called to deallocate a 4K page that was previously allocated with AllocatePage. The function must adhere to the following behaviour:

- No action should be taken if address is less than 0 or does not fall on a 4K boundary.
- If address is valid the 4K block of memory can be returned to the pool of available memory blocks.

```
PROCEDURE Dispose*(p: SYSTEM.PTR);
```

Description This procedure will be called whenever the memory previously allocated to dynamic structures using NEW must be deallocated. The function will adhere to the following behaviour:

- The function will dispose of the memory area pointed to by p and return the memory area to the pool of available memory blocks. The deallocated memory block will be merged with all adjacent blocks to create contiguous blocks. The memory block associated with p will be considered adjacent iff the address pointed to by p sequentially follows the end address of a free memory block or the start address of a free memory block sequentially follows the end address of the block pointed to by p.

- The procedure will take no action if p is equal to NIL (0).
- The procedure will take no action if p is a valid address, but the size of the memory block associated with p is less than or equal to 0.

```
PROCEDURE FreeDMA*(address: ADDRESS);
```

Description This procedure will be called whenever the memory previously allocated with AllocateDMA must be deallocated and must be presented with the start address of the DMA buffer. The procedure will adhere to the following behaviour:

- The procedure will dispose of the memory pointed to by address and return the memory area to the pool of available memory blocks. The deallocated memory block will be merged with all adjacent blocks to create contiguous blocks. The memory block associated with address will be considered adjacent iff its sequentially follows the end address of a free memory block or the start address of a free memory block sequentially follows the end address of the block pointed to by address.
- The procedure will take no action if address is equal to 0.
- The procedure will take no action if address is valid, but the size of the memory block associated with it is less than or equal to 0.

```
PROCEDURE Initialize*;
```

Description This procedure will be called by osInit and will perform the following initialization tasks:

- The procedure will determine the amount of memory that can be used by the various allocation routines and initialize the necessary data structures to support memory allocation and deallocation.
- The amount of lower memory available for allocation will start 4K above the kernel stack top and stop at physical address 9FFFFH. The btKernelStackTop field in the boot table can be used to determine the physical start address of the lower memory block.
- The amount of extended memory will be obtained from the boot table as follows:
  1. The btSMAPEntries field will be examined to determine the number of system memory map entries that were reported by the extended BIOS service 15H, function E820H. If a non-zero value is returned, the system will proceed by retrieving every system memory map entry using the GetSMAPEntry procedure to build a memory map and determine the amount of extended memory.
  2. If there are no system memory map entries, the system will proceed by examining the btExtendedMemoryE801 field to determine the amount of available memory above the 1MB limit.
  3. If the btExtendedMemoryE801 entry did not report any extended memory, the system will proceed by examining the btExtendedMemory entry. This entry may not report all the extended memory due to limitations in the various BIOS implementations. If the reported amount is either 15360 or 64512, the system will attempt to probe for additional memory.

- The procedure will allocate a page directory as well as an arbitrary number of page tables to map all physical memory, including the 384K region between physical address A0000H and FFFFFH. The first 4K (physical address 0H to FFFH) as well as the 4K above and below the kernel stack will be marked as not-present pages. All other entries in the page directory and page tables must be marked as read/write, present, supervisor pages.
- The procedure will enable the processor's paging mechanism once all the page directory and page table entries have been created.

4.1.1 Linking

The linker's parameters must be modified once osStorage is implemented and integrated into the system. A typical link operation will now look as follows:

```
BootLinker.Link os314
  \list osModules.modules
  \mdesc osModules.ModuleDesc
  \expdesc osModules.ExportDesc
  \new osStorage.NewRec \sysnew osStorage.NewSys \newarr osStorage.NewArr
  \integrate 1000H
  osModules osBootTable osTrace osMachine osStorage os8259
  osExceptions osTimer osInit ~
```

The \new, \sysnew and \newarr parameters instruct the linker to patch all references to NEW to the procedures in osStorage responsible for memory allocation. If the system contains any references to NEW and the additional parameters are not specified, the system will fail because the linker will by default patch the references to those procedures inside the Oberon Kernel module responsible for memory allocation.

4.2 Module: osDMA

This module must provide the necessary mechanisms to manage the allocation and deallocation of DMA channels as well as programming the DMA controller for both read and write operations. The module must support both 8-bit and 16-bit channels and the implementation must be designed for the Intel 8237 DMA controller.

```
PROCEDURE RequestChannel*(channel: LONGINT; VAR id: LONGINT): LONGINT;
```

Description This function will be called whenever a device driver requires access to a specific DMA channel. A unique identification will be assigned to the device driver to prohibit other devices from using the channel until it is released. The function must adhere to the following behaviour:

- The function will assign a unique value to id and return dmaOK if the operation was successful. The channel must be marked as allocated and removed from the list of available channels. The identification number associated with the channel must be recorded to verify further DMA requests involving the channel.
- The function will assign -1 to id and return dmaInvalidChannel if an illegal channel number was requested.
- The function will assign -1 to id and return dmaReservedChannel if a reserved channel number was requested.

- The function will assign -1 to id and return dmaChannelAllocated if the channel is already allocated.

```
PROCEDURE ReleaseChannel*(channel: LONGINT; id: LONGINT): LONGINT;
```

Description This function will be called whenever a device driver must deallocate a DMA channel that was previously allocated with RequestChannel. The identification number associated with the channel must be provided to prohibit other devices from releasing the channel. The function must adhere to the following behaviour:

- The function must return dmaOK if the operation was successful. The channel must be marked as available and placed back into the list of available channels.
- The function will return dmaInvalidChannel if an illegal channel number is specified.
- The function will return dmaReservedChannel if a reserved channel number is specified.
- The function will return dmaChannelReleased if the channel has already been released.
- The function will return dmaInvalidID if the identification number specified in id does not match the identification number that was recorded when the channel was allocated.

```
PROCEDURE Start*(channel, operation, address, length, id: LONGINT): LONGINT;
```

Description This function will be called whenever a device driver must perform an I/O operation to transfer data using DMA. The function must adhere to the following behaviour:

- The function will return dmaOK if all the parameters were verified and the requested operation was successfully programmed.
- The function will return dmaInvalidChannel if an illegal channel number is specified.
- The function will return dmaReservedChannel if a reserved channel number is specified.
- The function will return dmaChannelReleased if the channel is listed as an available channel.
- The function will return dmaInvalidBufferLength if the value in length exceeds the maximum buffer capacity associated with the specified channel or is less than or equal to 0.
- The function will return dmaInvalidOperation if the value in operation is illegal. Valid operations are: opIOToMemory and opMemoryToIO.
- The function will return dmaInvalidID if the identification number specified in id does not match the identification number that was recorded when the channel was allocated.

```
PROCEDURE GetStatus*(controller: LONGINT; VAR status: SET): LONGINT;
```

Description This function will read the status register of the specified controller to determine if a DMA operation completed successfully and whether or not there are any outstanding DMA requests. The function must adhere to the following behaviour:

- The function will read the specified DMA controller's status register and copy its contents to status and return dmaOK if the operation completed successfully.
- The function will set status to {} and return dmaInvalidController if an invalid DMA controller was specified. Valid controllers are dmaPrimary and dmaSecondary.

4.3 Module: `osProcess`

This module will implement the necessary abstractions to support concurrent processes and must support a number of primitives to perform various process related tasks. The module must provide the following minimum functionality:

- Functions to manipulate the state of a process.
- Functions to permanently remove a process and release all of its allocated resources.
- A scheduler that will select the next process for execution from a set of available processes.
- Data structures to implement *process control blocks* (PCBs) and the necessary routines to manipulate these PCBs. The structure must provide for efficient operations without limiting future extensions to the system.

4.4 Module: `osLoader`

This module will be responsible for loading the user processes from disk into memory and will contain a single procedure called `Initialize`. This module will depend on the data structures and functions provided by `osProcess` to correctly initialize the PCB of a process once it has been loaded from secondary storage.

```
PROCEDURE Initialize*;
```

Description This procedure must adhere to the following behaviour:

- Hardware interrupts must be enabled using the `STI` machine instruction to allow the floppy diskette driver and system clock to function properly while loading processes from secondary storage.
- The procedure will perform a read operation (starting at physical sector 400 and proceeding in multiples of 100, for example 400, 500, 600, etc.) to determine if a valid process is located on disk. Processes may therefore never exceed 100 sectors (51200 bytes). If a valid process can not be located, the function may disable hardware interrupts (using `CLI`) and abort the load operation to continue with other initialization tasks. A process will be considered valid if the first byte of the process image contains the value `E8H` and the link base field in the header of the statically linked image contains a value of `20000000H`.
- If a valid process has been identified, the procedure must determine the size of the process by examining the information stored in the header of the process image (link base and heap start address). The procedure will use this information to determine the number of disk operations required to load the complete image from disk and to calculate the amount of 4K pages necessary to hold the image in memory.
- The procedure will allocate a 4K page for the process page directory and copy the complete contents of the kernel page directory to the newly allocated page directory (refer to `osStorage.CopyKernelPageDirectory`).
- The procedure will allocate one or more page tables to store the process image. The number of page tables will be determined by the size of the process and its stack size (16K) plus two additional entries above and below the process stack to protect the process from stack overflows and underflows. Additional pages must also be allocated to hold the actual statically linked process image. All allocated pages, including those used to hold the page table(s), will be marked as present, read/write, user-level pages.

- The procedure will allocate a new PCB (refer to `osProcess`) to store all the information and will set the initial process state to *ready*. The procedure must also allocate and initialize a *task state segment* (TSS) for the newly created process to enable the hardware protection mechanisms (refer to `osMachine.InitTSS`). The procedure must also initialize the process state (refer to `osMachine.SetCPUState`).

4.5 Module: `osComms`

This module will provide the necessary mechanisms in the kernel to support user-level, indirect, synchronous interprocess communication based on the traditional client/server model. Processes will communicate with each other through ports. The system's implementation may limit the number of ports that can be created, but must be designed in such a way that extensions and modifications can be made without disrupting other portions of the system.
User-level processes will initiate communication operations through a predefined library called `IPC.Mod`. The library will create and initialize any parameters that may be necessary before initiating a system call. The system call will be handled by a single processing unit (`osSysCalls`) located inside the kernel. This unit will then dispatch all communication related system calls to `osComms`. There is no implementation definition for `osComms`. Instead, the module's functional requirements are indirectly described from the user's perspective by examining how the `IPC` library will react in various situations.
Care should be taken when transferring messages between processes. Since every process is isolated in a separate virtual address space, it may be necessary to first transfer a message from the sender to an internal kernel buffer before switching to the receiver's address space and transferring the message from the kernel buffer to the receiver. The same technique can be applied when a server sends a reply to a client process. However, this method does incur some overhead because the number of memory operations (read/write) are effectively doubled. This could pose a problem when large messages are transmitted between processes. An alternative would be to map the address space occupied by the message buffer of one process into the address space of the other process. This will eliminate the overhead of the first solution, but introduces additional complexities to ensure that addresses are correctly mapped. It should also be noted that translating the virtual addresses to physical addresses to reduce the amount of work to a single read/write cycle is not feasible because message buffers may span multiple pages and although the virtual address space is contiguous, the physical location of the pages associated with a process may not be contiguous.

4.5.1 Module: `IPC`

The `IPC` module provides a user-level library that can be accessed by processes to perform synchronous communication. Every procedure call to this library will result in the execution of a system call that will transfer control to the communications module (`osComms`) to handle the request. Refer to Table 2 for a detailed specification regarding the registers used to transfer specific parameters during a system call.

```
PROCEDURE CreatePort(port: LONGINT): LONGINT;
```

Description This function will typically be called by a server to create a communications port that can be used to process requests from other processes and will adhere to the following behaviour:

- The function will return `ipcOK` if the requested port could be created.
- The function will return `ipcDuplicatePort` if the requested port already exists.

Procedure	Parameters					
	EAX	EBX	ECX	EDX	ESI	EDI
CreatePort	ipcCreatePort	port				
LookupPort	ipcLookupPort	port				
Send	ipcSend	port	reqbuf	reqlen	repbuf	replen
Receive	ipcReceive	port	reqbuf	reqlen		
Reply	ipcReply	port	repbuf	replen		
ClosePort	ipcClosePort	port				

Table 2: Parameter assignments for IPC system calls.

- The function will return `ipcInvalidPort` if the requested port number is less than 0 or greater than the maximum port number supported by `osComms`.

```
PROCEDURE LookupPort(port: LONGINT): LONGINT;
```

Description This function will typically be called by a client to determine if a given server is active and will use the port to send requests to the server that owns the port. The function will adhere to the following behaviour:

- The function will return `ipcOK` if the requested port could be located.
- The function will return `ipcInvalidPort` if the requested port number is less than 0 or greater than the maximum port number supported by `osComms`.
- The function will return `ipcNoSuchPort` if the requested port does not exist, thereby indicating that the server has not yet created the port.

```
PROCEDURE Send(port: LONGINT; VAR reqbuf: ARRAY OF SYSTEM.BYTE; reqlen: LONGINT;
               VAR repbuf: ARRAY OF SYSTEM.BYTE; replen: LONGINT): LONGINT;
```

Description This function will be called by a client to send a message to a server and will adhere to the following behaviour:

- The function will return `ipcOK` if the server could receive the message, process the request and respond with a valid reply.
- The function will immediately return `ipcNoSuchPort` if the requested port does not exist and the client process will not become blocked.
- The function will immediately return `ipcInvalidPort` if the requested port number is less than 0 or greater than the maximum port number supported by `osComms`. The client process will not become blocked.
- The function will immediately return `ipcInvalidMessageSize` if either `reqlen` or `replen` is less than 1 or greater than the maximum message size supported by `osComms`. The client process will not become blocked and the server will not receive the message.
- The function will immediately return `ipcInvalidBuffer` if the address of the message buffer (`reqbuf` or `repbuf`) is less than or equal to 0.

At this point `osComms` will block the client (sender) and will attempt to transfer the message to the server (receiver). If the server is busy, the message will be placed in a queue and processing of the message will be postponed until the server is ready to accept new requests. If the server process executed a `Receive` system call and is currently blocked with no outstanding messages, then the message will immediately be transferred to the server and the server will become unblocked. The client process will only become ready once the server successfully received the message, processed the transaction and successfully send a reply. Any errors that occur during these steps will be handled by `osComms` and the return value of `Send` must be updated to reflect these results to allow the client to respond correctly once it becomes ready (unblocked).

```
PROCEDURE Receive(port: LONGINT; VAR reqbuf: ARRAY OF SYSTEM.BYTE;
                  reqlen: LONGINT): LONGINT;
```

Description This function will typically be called by a server to either wait for new messages or process outstanding requests that may have been queued on the communications port while it was busy processing other requests. The function will adhere to the following behaviour:

- The server (receiver) will become blocked if a valid communications port (`port`), valid buffer (`reqbuf`) and valid buffer size (`reqlen`) were specified and there are no messages waiting on the port's message queue. If the parameters are valid and the port contains outstanding messages, the server will not be blocked. Instead, the first message will be transferred and the server will immediately begin to process the message and the function will return `ipcOK`. The server will become blocked if there are no outstanding messages to receive.
- The function will immediately return `ipcNoSuchPort` if the requested port does not exist and the server process will not become blocked.
- The function will immediately return `ipcInvalidPort` if the requested port number is less than 0 or greater than the maximum port number supported by `osComms`. The server process will not become blocked.
- The function will immediately return `ipcInvalidMessageSize` if `reqlen` is less than 0 or greater than the maximum message size supported by `osComms`. The server process will not become blocked.
- The function will immediately return `ipcInvalidBuffer` if the address of the message buffer (`reqbuf`) is less than or equal to 0.

```
PROCEDURE Reply(port: LONGINT; VAR repbuf: ARRAY OF SYSTEM.BYTE;
                replen: LONGINT): LONGINT;
```

Description This function will typically be called by a server once a request has been successfully received and processed. The function is used to send a reply back to the client (sender) and completes the communication cycle. The function will adhere to the following behaviour:

- The function will return `ipcOK` if the message could successfully be send to the client process to complete the communication cycle. The server will not become blocked when it executes this function.
- The function will return `ipcNoSuchPort` if the requested port does not exist. The message will be removed from the message queue and the client that executed the `Send` operation will become unblocked and `ipcNoSuchPort` will be returned as the result for the `Send` function.

- The function will return `ipcInvalidPort` if the requested port number is less than 0 or greater than the maximum port number supported by `osComms`. The client that executed the `Send` system call will become unblocked and will receive `ipcInvalidPort` as a function result.
- The function will return `ipcInvalidMessageSize` if `replen` is less than 0 or greater than the maximum message size supported by `osComms`. The client that executed the `Send` system call will become unblocked and will also receive `ipcInvalidMessageSize` as a function result.
- The function will return `ipcInvalidBuffer` if the address of the message buffer (`repbuf`) is less than or equal to 0. The client that executed the `Send` system call will become unblocked and will also receive `ipcInvalidBuffer` as a function result.

```
PROCEDURE ClosePort(port: LONGINT): LONGINT;
```

Description This function will typically be called by a server when it terminates. The function must adhere to the following behaviour:

- The function will return `ipcOK` if the port could successfully be closed.
- The function will return `ipcNoSuchPort` if the specified port does not exist.
- The function will return `ipcInvalidPort` if the specified port number is less than 0 or greater than the maximum port number supported by `osComms`.
- Any outstanding messages in the port's message queue will be removed and the various `Send` system calls will receive `ipcPortClosed` as a result after the clients have been unblocked.

4.5.2 Communication Ports

Communication ports provide an abstraction of the communication mechanism and allows processes to communicate with each other without the need to directly name the target process. The ports must provide a mechanism to queue an arbitrary number of messages in the event that a server receives additional requests from clients before it has completed servicing its current request.

4.5.3 Synchronous Communication and Process Behaviour

A typical communications scenario is illustrated in Figure 2. The system must adhere to the following behaviour:

- The server will be responsible for creating a communications port using the `IPC.CreatePort` system call.
- The client will poll the system repeatedly using the `IPC.LookupPort` system call until it receives `ipcOK` as a result, indicating that the port was located. The client may take any additional actions deemed necessary should it receive a result other than `ipcOK`. The system may return `ipcNoSuchPort` depending on the process inter-leavings and the client should attempt multiple calls if this result is received because the client may be scheduled before the server, prohibiting the server from creating the port being searched for by the client.
- The server will typically enter an endless loop to await requests from client processes after creating the communications port. The server will execute `IPC.Receive` to await a request and will be blocked until a valid client request is received through the communications port (Step 1).

- The client process will send a request to the server by executing `IPC.Send` and will block until a reply from the server is received (Step 2).
- The server will be unblocked once a request is received (Step 3) and will process the request before sending a reply by executing `IPC.Reply` (Step 4) and returns to the start of the transaction loop (Step 5). It is possible that the server may receive additional requests while processing the current request before it is able to execute `IPC.Receive` again. In this case, the clients will be blocked by the system and their requests must be queued by the communications mechanisms inside `osComms`.
- The client process will be unblocked once a reply is received (Step 6).

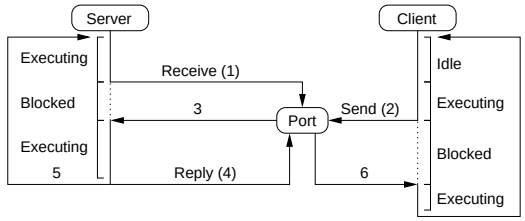

Figure 2: A typical communication cycle using IPC.

4.6 System Calls

All system calls will be dispatched to `osSysCalls` and will be invoked by executing an `INT 0FFH` machine instruction. Table 3 lists all the system call function numbers. Upon receiving a request, `osSysCalls` will either call the appropriate module inside the kernel responsible for the requested service or return control back to the user-level process if the system call is not recognized. The requested service routine will determine whether or not the kernel must perform a rescheduling operation to select a new process once the system call has been completed. A typical scenario is illustrated in Figure 3.

- Process 0 is in a running state and executes a system call provided by some library (`TTY`, `IPC`, etc.). The library will initialize the necessary parameters before executing a software interrupt machine instruction that will transfer control to `os8259.FieldInterrupt` through the IDT (Step 1).
- The system call handler (`osSysCalls.RequestHandler`) is called by `FieldInterrupt` (Step 2).
- The handler will examine the requested service and call the appropriate procedure inside the kernel (Step 3).
- Every system call will indicate whether or not a rescheduling operation is required and `osSysCalls.RequestHandler` will pass this information back to `FieldInterrupt` (Step 4).
- At this point, control is either transferred back to Process 0, or the scheduler is invoked to select a new process (Process 0 ... Process n) as illustrated in Step 5.

Module	System Call	Function
TTY	Open	00H (0)
	Close	01H (1)
	Clear	02H (2)
	GotoXY	03H (3)
	Ln	04H (4)
	Char	05H (5)
	String	06H (6)
	Int	07H (7)
	Hex	08H (8)
	Bin	09H (9)
	Bit	0AH (10)
	Mem	0BH (11)
	SetFgCol	0CH (12)
	SetBgCol	0DH (13)
	KeyAvailable	64H (100)
	ReadKey	65H (101)
IPC	CreatePort	C8H (200)
	Send	C9H (201)
	Receive	CAH (202)
	Reply	CBH (203)
	Lookup	CCH (204)
	ClosePort	CDH (205)
SYS	Exit	FAH (250)
	GetTicks	FBH (251)

Table 3: System call summary.

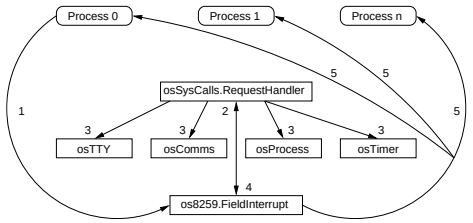

Figure 3: Control transfer during a system call.

5 Processes

5.1 Implementation

All user level processes will be developed in Oberon using the OP2 Oberon compiler and static linker. Processes may only import standard libraries such as `TTY`, `IPC` and `SYS`, or other Oberon independent libraries. All user processes will be linked against virtual address `20000000H` as illustrated in the example below.

```
BootLinker.Link myprocess
  \list osModules.modules
  \mdesc osModules.ModuleDesc
  \expdesc osModules.ExportDesc
  \integrate 20000000H
  osModules IPC TTY MyProcess ~
```

5.2 Process Management and Behaviour

The system will use the internal 8253 timer chip to schedule processes and will allocate a time slice of no more than 20ms to every process. A process may only be scheduled if it is in the *ready* state and only one process may be in a *running* state at any given time. Exceptions generated through non-conforming behaviour of processes will result in the system immediately terminating the process (note that a process may volunteer for termination by executing the `SYS.Exit` system call). Termination implies that the process will be removed from the scheduling queue and all allocated memory must be returned to the operating system. Once the process has been removed, the system must resume its normal operation by simply scheduling a new process. The system must still service interrupts even if there are no processes or no process indicates that it is ready for scheduling. Extensions must be made to `os8259.FieldInterrupt` to call the scheduler, dispatch processes or terminate erroneous processes.

6 Testing

It is the sole responsibility of the system developer to thoroughly test all the modules by applying different testing techniques such as functional, boundary and integration testing. Coverage analysis (multiple condition and branch coverage) of the modules and the system as a whole will not be required.