Chapter 8

Nut Operating System

In this chapter we develop Nut operating system (Nos). This operating system runs on the Sx processor. Nos is a preemptive multitask operating system. This operating system has many services: interprocess communication, shared resources, process synchronisation, and a real time clock. As we have the processor simulator, the implementation of interrupt and task switching can be studied in details down to the level of machine cycle. A supervisor program (Nos supervisor) is created to interface between Nos and the Sx processor simulator. Nos is designed to be very limited. It does not have virtual memory, the file system nor the network. Although it is very limited, it does offer an insight into the essential part of operating systems.

8.1 Operating system concepts

A process is a basic unit of abstraction to build concurrent execution of multiple programs. A process is a program in execution. A program is *static* whereas a process is *dynamic*. One program can be executed by many processes. A process consists of: code segment, data segment, stack segment. They can be shared or separated. When they are shared, only a single address space is needed hence the implementation is simple.

To achieve concurrency using one processor, each process will be allocated a slice of time for its execution. All processes will be scheduled to be executed by time multiplexing. For example, two processes *A* and *B*, to run concurrently they will be executed like this:

A B A B A B A B

A simple programming abstraction to achieve this is *co-routine* where *A* calls *B* then *B* calls *A* but not starting *A* at the beginning. *A* resumes the execution at the point where it has previously stopped.

Several processes can be active at the same time. The concurrency is achieved via multi-threading (light weight process). A light weight process has single address space. A heavy weight process is a process with a separate address space. It needs a mechanism to do the address mapping between virtual address and physical address. A thread is a trace of execution. Concurrency with a single thread process is achieved by co-operative process (via co-routine). A multi-thread process has several traces at the same time. This can be accomplished by pre-emptive scheduler with time-slicing. A light weight process is much cheaper to create than a heavy weight process.

To manage multiple programs, a scheduler keeps a list of all processes. When starting an execution of a time slice, a process will be selected to be run according to some policy. A process is run until it is:

1. time-out using up its allotted time.
2. blocked the process requests a resource and has to wait for it.
3. terminated the process runs to its completion.

We can model the behaviour of the process as a state machine as shown in Table 8.1.

Table 8.1 The state of a process

State	Event	Next state
READY	Task switch	RUNNING
RUNNING	Time-out	READY
RUNNING	Terminate	DEAD
RUNNING	Block	WAIT
WAIT	Wakeup	READY

To execute a process on a processor, the computation state (C-state), i.e. all variables pertain to that process must be saved/restored properly upon a task switching. These are PC, FP, SP, and TS including separate stack segment for

each process. These values are stored in a data structure called the *process descriptor*. The task-switch is defined as follows.

```
task-switch
   if only one process do nothing
   save current state, set it to READY
   select next process
   make it runnable, set it to RUNNING
```

To understand how a process requests a resource we first study how a resource is shared.

In sharing a resource, it is necessary to ensure *mutual exclusion*. That is, during a period of one process possessing that resource, other process that also wants that resource must wait. Dijkstra invented "semaphore" [DIJ65] to achieve this "mutex" behaviour. A semaphore is a variable (can be binary, 0/1, or an integer) associated with a resource. It indicates availability of the resource to the process that requested it. Two operations are defined on a semaphore: wait, signal (originally Dijkstra called it P, V). Associated with each semaphore is a waiting list of the process that waits on this semaphore. They are defined as follows:

```
Wait sem
   if value of sem <= 0
      block this process, set it to WAIT
      put this process to waiting list of
      this sem
  else
      sem - 1

Signal sem
   if there is process waiting for this sem
      move that process out of waiting list
      wakeup that process
   else
      sem + 1
```

These two operations must be atomic, that is, they must run to completion without interruption (can not task switch in between).

For resource sharing, we let only one process to acquire the resource, all other processes that request that resource will be waiting in the waiting list. The code where this mutual exclusion is required is said to be *critical section*. A semaphore is used to protect this section. Start with `sem` = 1.

```
....
; critical section
wait sem
... code to access shared resource
signal sem
```

The first process that reaches this section will "close the door". After it finishes with this section, it "open the door" to let other process in. A semaphore is the basis that other mechanism can be built on such as process synchronisation or interprocess communication.

Two processes need to be *synchronised* at some point in the program. Two semaphores are used to achieve it. Let two processes be *A* and *B*. Assume *A* arrives to the synchronise point before *B*. Then *A* must wait for *B* and vice versa if *B* arrives before *A*.

```
    A                   B

signal sem1         wait sem1
wait sem2           signal sem2
```

The sequence can be rearranged and the behaviour is still correct.

```
   A                    B

signal sem1         signal sem2
wait sem2           wait sem1
```

The interprocess communication is achieved using synchronous message-passing. It combines communication and synchronisation in a single high-level primitive. Other alternative models for message-based process synchronisation are: asynchronous and remote invocation.

There are relationships between asynchronous, synchronous and remote invocation semantics. Two asynchronous events can constitute a synchronous relationship if an acknowledgement message is always sent (and waited for). Two synchronous communications can be used to construct a remote invocation. It could be argued that the asynchronous model gives the greatest flexibility but there are a number of drawbacks:

1. Potentially infinite buffers are needed to store messages that have not been read yet.
2. In asynchronous model, more communication are needed, hence programs are more complex.

Also, a synchronous model can emulate an asynchronous communication simply by using a buffer process.

A system is said to be *hard real-time* if it has deadlines that cannot be missed for if they are, the system fails [LEI80]. A system is *soft real-time* if the application tolerant of missed deadlines. A system is *interactive* if it does not have specified deadlines but strives for adequate response times.

Two types of process are present in the real-time domain: periodic and aperiodic. Periodic processes sample data or execute a control loop and have explicit deadlines that must be met. Aperiodic processes (or sporadic) arise from external asynchronous events. These processes have specified response time associated with them. The process must be analysed to give its worst-case execution time, also may obtained average execution time [BUR01].

To schedule real-time tasks, "schedulability" is an important concept. Given a collection of processes and all associated deadlines, determine if this set of processes is schedulable. This means that it is possible for all deadlines to be met indefinitely into the future. In general, necessary and sufficient conditions for schedulability are not known. However, there are many different algorithms presented in the literature which test for schedulability under certain preconditions and restrictions [SHA88] [LEH89].

8.2 Nut operating system

Nut operating system (Nos) is written in Nut. It runs as a user program. Nos models concurrency using process. The share-resource accesses are controlled with semaphores. The processes in Nos communicate with each other through message passing. The crucial real-time functions are supported. Therefore Nos can support real-time tasks. Nos supports the following functions:

- create a process
- terminate a process
- manage the process queue
- task switching

- wait/signal a semaphore
- send/receive a message
- get a real-time clock
- set a timer

8.3 Process

A process is an independent computation which can run concurrently with other process. A process is declared as a normal defined function. Initial values can be passed as parameters at the starting time of a process. A process will end its execution by self-termination when it executes the last instruction at the end of program. This is different from the execution of a function which ends its execution by returning to the caller. A program that calls a process will start that process execution, that program then will continue to work without waiting. A process never returns to its caller.

Each process has its own stack segment. In Nos, there is a single address space, the stack segment of all processes are in the same address space. The advantage is that there is no translation between virtual address and physical address therefore it is fast and simple. The disadvantage is that there is no protection between processes. Each process has its process descriptor (PD) to store the necessary information. A process descriptor in Nos consists of 12 fields.

1. link previous
2. link next
3. process id
4. process status
5. PC
6. SP
7. FP
8. TS
9. in-box
10. await-box
11. message
12. timer

The links previous and next are used to form the list of processes, for example the process queue. The process identifier is a unique number used to label a process. The process status holds the state of process (to be explained later). The fields {PC, SP, FP, TS} hold the computation state of the process. The mail-box (in-box and await-box) is used to communicate between processes. The in-box is the list of processes that sent messages to this process. The await-box is the list of

processes that are waiting for messages from this process. The timer holds the timer value of this process.

8.4 Scheduler

When a process is created it is ready to start the execution. Its PD will be linked to the *ready list* (the process queue) which is a doubly linked circular list used by the scheduler. A scheduler has the duty of selecting a process to run from the ready list. The scheduling policy of Nos is a Round-Robin policy where an equal time-slice is allocated for every process and the process is scheduled on first-come first-serve basis. A scheduler will enable a process in the ready list to run until its time-slice is over and then switches to the next process in the list. If a process enters a *wait* state, it is said to be blocked. A process is usually blocked because it performs some operation that requires waiting for another process, such as waiting for the receiver to retrieve a message. When a process is blocked, its PD will be removed from the ready list. The process in *wait* state can be awaken by other process. Its PD will be inserted into the end of the ready list. To perform the switch from one process to another process (task switching), the current state of computation {PC, SP, FP, TS} of the active process is saved in its PD and the state of computation of the process to be run is restored. The first process to be active is the process to run the main program. A process is run until it is time-out, or it is blocked or it is terminated. "switchp" is the task switcher function in Nos. Here is how "switchp" work.

```
switchp
  if status is time-out or blocked
    runnable next task
  else status is terminated
    delete this task
    runnable next task
```

Where status is the state of the process, runnable marks the process as running. Here is the actual code in Nut.

```
(def switchp () ()  [nos 127]
   (do
   (di)
   (if (or (= status TIMEOUT) (= status LOCKED))
      (do
      (setValue activep READY)
```

```
(set activep (getNext activep))      ; switch next
(runnable activep))
; else                               ; status STOPPED
(do
(setValue activep DEAD)
(set activep (deleteDL activep))
(if (!= activep 0)
   (runnable activep))))))
```

Before we go into more details of the operating system functions, we must understand the basic of running a task first.

8.5 Nos supervisor (Noss)

Nos is executed under Nos supervisor (Noss). Nos supervisor runs on top of the processor simulator. Noss is a privileged program. A privileged program is a program that is "out-of-bound" of user programs. A privileged program provides mechanism for executing a user program, for example, the Sx processor simulator is a privileged program that *executes* S-code. In our implementation, privileged programs are written in C. User programs are written in Nut. The relationship between Nos supervisor and the processor simulator can be understood by regarding Noss as issuing an *interrupt* to the processor.

The processor executes a program continuously (running a process) until an interrupt occurs then the supervisor takes action. The interrupt can occur only at the end of executing an instruction. There are three interrupt events: time-out, stop, and block.

time-out − the current process has used up its own time-slice.
stop − the current process runs to completion.
block − the current process is blocked (to be resumed later).

The supervisor (Noss) takes action in response to the interrupt events as follows.

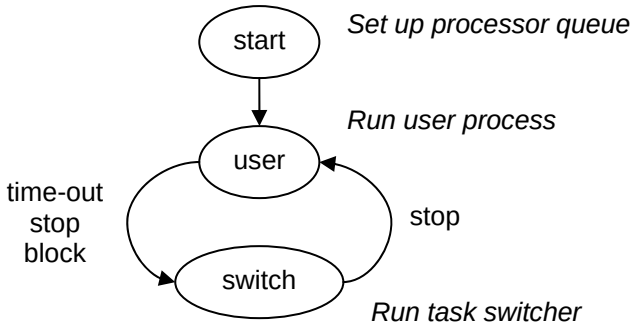

Figure 8.1 Noss state diagram

At start, the main program creates processes and the process queue. Noss schedules only two kinds of processes: user process and switchp. "switchp" is created and has its own PD but it is privileged and never enters the process queue. Once the process queue is ready, at the state user, a user process is run. The user process is run until it is time-out/stopped/blocked. Then Noss calls the "switchp", at the state switch. "switchp" runs to completion. It must not be interrupted as it is manipulating the process queue. Then the state is going back to run a user process.

At the end of task-switch, Noss always runs the next task. This is accomplished by restoring the computation state (C-state) of that process. This means the PC, SP, FP, TS of that process are restored to the processor simulator and then the processor simulator continues to execute until the interrupt occurs.

The process descriptor contains C-state. Saving and restoring C-state are the act of transferring C-state between the processor simulator and the process descriptor. Noss does the restoring of C-state. This restoring will affect the flow of execution, as the instruction pointer is changed; it does a jump in the program. As Noss is responsible to run the task-switcher, it must save C-state. Saving C-state can not be done in user-space as the precise state has been changed when trying to run the "saving state" function. So, save-C-state is done in the main loop of Noss (in C). This gives the save-C-state a special privilege (so called *kernel* in OS vocabulary). The following pseudo code described Noss.

```
Noss  [noss 43]
  if there is no process in the queue
     stop the whole simulation
  else
     update status to nos
     if state = switch
        save current user process
        restore "switchp"
        state = user
     else state = user
        restore active process
        state = switch
```

8.6 Simulation of interrupt

The processor simulator always runs a program in a tight loop. The processor fetches an instruction and executes it. To simulate an interrupt, the processor calls Noss from time to time (this is called *yield*). The interrupt interval is controlled by counting the cycle used since the last call to Noss.

The processor returns the control back to Noss after three conditions:

1. Its time-slice has been used up. This is called "time-out".
2. The process has been blocked by executing some operation. This is called "blocked".
3. The task is completed. The program reached "end". This is called "stopped".

When the processor hands the control back to Noss, Noss calls task-switch. The task-switch code is in the user-space. The task-switch is the function "switchp". "switchp" requires the knowledge of the status of the completion of the previous task: time-out, blocked, or ended. This information is provided by Noss via the variable "status", as Noss controls the processor simulator it knows how the task has returned the control.

Noss is minimal in the sense that it does not do a lot of things by itself. The only thing it does is to call "switchp". Noss monitors the state of computation of a process through two global variables: status and activep. activep points to the current process descriptor.

8.7 Processor simulator

In the processor simulator, the main simulation loop is "eval". It executes a fixed number of cycles. This is the main fetch-execute cycle of the processor (in fact most processor simulators are like this):

```
eval
  count = 0
  loop
    if count > limit break
    fetch an instruction
    execute the instruction
    count = count + 1
```

To implement interrupts, a flag (intflag) is used to disable the break. This flag can be turned on/off by the system calls.

```
eval2
  count = 0
  loop
    if intflag == 1
      if count > limit break
    fetch an instruction
    execute the instruction
    count = count + 1
```

The system calls that support Nos are:

20 disable interrupt
21 enable interrupt
22 block a process

The following code describes the main loop of the processor simulator.

```
eval  [noss 71]
  set PC
  while runflag = 1
    run one instruction
    yield

yield
  if intflag = 1
    if no of cycle used > TIMEOUT
       noss(time_out)
```

Where runflag controls the termination of Noss itself, intflag is the interrupt flag used to disable/enable interrupt.

8.8 How a process is created

A function "run" is used to create a process and put it to the process queue. Although "run" looks like a normal function, it can not be compiled into a normal function call. The argument to "run" is a function which will be turned into a process so it should not be evaluated. A call to "run" is compiled into the code that passes an address of the function. The argument to "call.run" in N-code is just a user-function call with its arguments as usual. However, this argument will not be evaluated. Instead, the address of the code of this call will be generated as an argument of "run". This code will be activated by the scheduler as a process. See the following example:

```
(def add (a b) () (+ a b))
(def run (f) () 0)
(def main () ()
   (do
   (run (add 4 5))))

add
(fun.2.2 (+ get.1 get.2 ))
run
(fun.1.1 (lit.0 ))
main
(fun.0.0 (do (call.17 (call.14 lit.4 lit.5 ))))
```

The generate S-code is as follows. See line 12-18.

```
 1 Call main
 2 End
 3 Fun add
 4 Get 2
 5 Get 1
 6 Add
 7 Ret 3
 8 Fun run
 9 Lit 0
10 Ret 2
11 Fun main
```

```
12 Lit 15
13 Call run
14 Jmp 19
15 Lit 4
16 Lit 5
17 Call add
18 End
19 Ret 1
```

The line "(run (add 4 5))" becomes

```
12 Lit 15       address of code (add 4 5)
13 Call run
14 Jmp 19       do not execute now
15 Lit 4        the code (call.add lit.4 lit.5)
16 Lit 5
17 Call add
18 End
19 . . .
```

8.9 How to generate code for run

The S-object must contain the symbol table with the correct references. The S-object is generated by "gen.txt". However, "gen.txt" just passes the symbol table through. The symbol table is read from N-object. The N-object is generated by "nut.txt", the compiler. The current version dumps everything in the symbol table.

The symbols that must be exported are of type FUN and GVAR only. The following tasks must be done.

1. Change "nut.txt" to output only the necessary symbols.
2. Change "gen.txt" to output the S-code reference. However the number of symbol does not change.

at nut.txt

The dumpsym [nut 104] is responsible to output the symbol table. relocate the reference to function such that the code segment starts at 2. It is necessary as nut-compiler is used under "nvm" where both the compiler and the user program to be compiled occupy the same code segment. Therefore the user program in

the code segment will not start at 2. We would like the object to be relocatable, therefore the user code should start at 2.

When starting the compiler, (sys 9) is used to find out where to user code segment is. The global variable "Start" stores this location, and it is used to relocate the reference to all function calls when output the object. The following code is added in "nut.txt" at dumpsym, to output the correct reference. The data segment is not relocated as it is already started at 0.

```
(if (= ty tyFUN)
   (set n (shift (getVal i) Start))
   ; else
   (set n (getVal i)))
(print n) (space)
```

at gen.txt

Here is the added code to outsym, [gen ?] to output the symbol table.

```
(set ty (atoi tok))
(tokenise)                ; ref, reloc
(if (= ty tyFUN)
   (do
   (set ref (shift (atoi tok) CS))
   (print (assoc ref)))
   ; else
   (prstr tok))
  (space)
```

When reading the symbol table from N-object, the generator recognises the type "fun" and outputs the S-address corresponding to the N-address.

To generate the code for the expression (run (fn ...)), the code to call "run" is generated and the address pointed to (fn ...) is generated as its argument.

```
   lit x
   call run
   jmp y
x: ...
   call fn
   end
y: ...
```

The address of *x* is at the next 3 words. *jmp y* skips the code (fn...). The call to (fn ...) is deferred and "*run*" will use *x* as the starting address of the process which calls (fn...). When the process returned, it will be terminated by "*end*". This is in the function gencall.

```
; convert arg to index to symtab
; e is arglist
(def gencall (arg e) (idx a)
   (do
   (set idx (searchRef arg))
   (if (= idx Runidx)        ; is "run"
      (do
      (outa icLit (+ XP 3))          ; point to code of process
      (outa icCall idx)              ; call run
      (set a XP)
      (outa icJmp 0)
      (eval (head e))
      (outs icEnd)
      (patch a (- XP a)))            ; jump over
       ; else
      (do                            ; normal call
      (while e
         (do
         (eval (head e))
         (set e (tail e))))
    (outa icCall idx)))))
```

Example session

The following example shows how a process is created and run. A user program is written as "(count n)"and integrated with Nos in "main":

```
; ---- application --------

(def count (n) (i)
   (do
   (set i 0)
   (while (< i n)
      (do
      (set i (+ i 1))
      (print i) (space)))))
```

```
(def main () (p)
   (do
   (sys 5)
   (set activep 0)                ; clear task-list
   (set sseg 1000)                ; allocate SS
   (set p (run (count 500)))      ; create "(count 500)"
   (bootnos)))
```

The line (run (count 500)) creates a process to run (count 500). (bootnos) starts the process running.

To run NOSS, first compile user functions with NOS in nos.txt:

```
e:\test>nut < nos.txt
```

Then run NOSS with the executable, let it be "a.obj".

```
e:\test>noss a.obj
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
38 39 40 41 42 43 44 45 46 47 48 49
*
50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 74 75 76
. . .
*
487 488 489 490 491 492 493 494 495 496 497 498
499 500
*
9345 clocks

e:\test>
```

The "*" indicates the task-switching (every 1000 cycles).

8.10 Interprocess communication

Nos provides two ways to communicate (passing some values) between processes:

1 by share variables
2 by message passing

Share variables

A semaphore is used to provide mutual exclusion of access to share variables. The share variable will be accessed by only one process at a time. (Remember that processes can be concurrent therefore at any time there can be more than one process trying to access the same variable). A semaphore is implemented as a special global variable with two fields: value, wait-list. The access to a semaphore is done via two functions: signal, wait. They are atomic operations. The operation runs to completion without interrupt. This is achieved by *disable interrupt* at the beginning of the function and *enables interrupt* before return.

```
; semaphore field: sval(value) slist(wait-list)

(def signal (s) (p)  [nos 166]
   (do
   (di)
   (set p (getslist s))
   (if (!= p 0)
      (do
      (setslist s (deleteDL p))
      (wakeup p))
       ; else
      (setsval s (+ (getsval s) 1)))
   (ei)))

(def wait (s) (v p)  [nos 178]
   (do
   (di)
   (set v (getsval s))
   (if (<= v 0)
      (do                                   ; block activep to WAIT
      (set p activep)
```

```
    (set activep (deleteDL activep))
    (setValue p WAIT)                    ; to wait-list
    (setslist s (appendDL (getslist s) p))
    (blockp))                            ; block
     ; else
    (setsval s (- v 1)))
  (ei)))
```

Where "blockp" blocks the current process (and calls the supervisor), "wakeup" puts the process p in the process queue, ready to be scheduled to run.

```
(def initsem (v) (s1)  [nos154]
  (do
  (set s1 (new 2))
  (setsval s1 v)
  (setslist s1 0)                        ; wait-list nil
  s1))

(def wakeup (p) ()  [nos 161]
  (do
  (setValue p READY)
  (set activep (appendDL activep p))))
```

A "monitor" can be constructed to provide an abstract data type to protect shared variables. The access is done via the parameter "cmd". The monitor uses the associated semaphore to perform *mutual exclusion* access. Only one process can be inside the monitor at one time.

```
(def monitor (cmd) ()
  (do
  (wait sem1)
  (if (= cmd 1)
    ... access shared variables
    ; else
    ... access shared variables
  (signal sem1)))
```

8.11 Message passing

The message passing in Nos is implemented as a blocking protocol where the sender and receiver wait until the exchange is completed before continuing. This

is done using two mail-boxes: in-box and await-box. Here is the pseudo code for the "send" and "receive" operations.

```
send p mess
  if there is a process p wait for it
    put mess to p's buffer
    wakeup p
  else
    block itself
    append itself to p's in-box

receive p
  if there is a process p mail in in-box
    take the message from p's buffer
    wakeup p
  else
    block itself
    append itself to p's await-box
```

The Nut program implementing "send" and "receive" is as follows.

```
; p is pointer to process
(def send (p mess) (m box)  [nos 221]
   (do
   (di)
   (set box (getAwait activep))
   (set m (findmail p box))
   (if (= m 0)
      (do
      (set m activep)              ; self
      (setMsg m mess)
      (set activep (deleteDL activep))
      (setMbox p (appendDL (getMbox p) m))
      (setValue m SEND)
      (blockp))
      ; else
      (do                          ; p is waiting
      (setMsg p mess)
      (set m (deleteDL p))
      (if (= box p)
         (setAwait activep m))
   (wakeup p)))
   (ei)))
```

```
(def receive (p) (m box)  [nos 243]
   (do
   (di)
   (set box (getMbox activep))
   (set m (findmail p box))
   (if (= m 0)
      (do                          ; put to await p
      (set m activep)              ; self
      (set activep (deleteDL activep))
      (setAwait p (appendDL (getAwait p) m))
      (setValue m RECEIVE)
      (blockp)
      (getMsg m))                  ; retrieve from self
      ; else
      (do                          ; already in mbox
      (set m (deleteDL p))
      (if (= box p)
         (setMbox activep m))
      (getMsg p)                   ; retrieve mbox
      (wakeup p)))
   (ei)))
```

There are two buffers, one in the sender and other in the receiver. The process descriptor is attached to the in-box/await-box so that waking up a process associated with the mail is simple. "findmail" searches for a message from a process p in the mail-box. "blockp" blocks the current process (and calls the supervisor). The state "SEND/RECEIVE" indicate that the process is blocked by the send/receive operation. "wakeup" puts the process p in the process queue, ready to be scheduled to run.

Example of send/receive message

We write two functions, one is the producer that sends the message *2..n,* the other is the consumer. The consumer receives the message until the end of message is reached (-1).

```
(def produce (n) (i)  [nos 270]
   (do
   (set i 2)
   (while (< i n)
      (do
      (send p2 i)
      (set i (+ i 1))))
```

```
   (send p2 (- 0 1))))

; receive 2..n from p1 ended with -1
(def consume () (m flag)  [nos 281]
   (do
   (set flag 1)
   (while flag
      (do
      (set m (receive p1))
      (if (< m 0)
         (set flag 0))))
   (nl)))
```

Create and run producer and consumer.

```
(def main () ()  [nos 292]
   (do
   (di)                          ; disable interrupt
   (set activep 0)               ; init task-list
   (set sseg 1000)               ; init stack segment
   (set pid 1)                   ; init process id
   (set psw (run (switchp)))     ; create switchp
   (set activep 0)               ; clear task-list
   (set p1 (run (produce 1000)))
   (set p2 (run (consume)))
   (bootnos)))
```

When running the above program the producer streams the messages (integers) *2..n* to a consumer. The producer's output is marked "!n" and the receiver's output is marked " "n". The task-switched is marked "*". The trace is:

```
!2  * "2  * !3 !4  * "3 "4  * !5 !6  * "5 "6  * !7 !8
* "7 "8  * !9  *  "9 . . .
```

Let two processes be s, r. This behaviour can be explained by inspecting the trace of execution of two processes.

notation

sM send in-box
sA send await
rM receive in-box

rA receive await
sB sender block
rB receiver block

The trace is:

```
1 producer: sM sB *
2 consumer: rM rA rB *
3 producer: sA sM sB *
4 consumer: rM rA rB *
. . .
```

The first line says that the sender just sent a message to the receiver's in-box then itself is blocked. The second line is quite interesting. It says the receiver retrieves the message from the sender's buffer and then continues to execute its program which does "receive p". This call forces the receiver to send itself to the sender's await-box, and then itself is blocked. This mean "r" is waiting for a message from "s". Once "s" wakeups "r", the process "r" will have a message in its buffer. Line 3, 4 can be similarly explained.

This benchmark has been compiled with all optimisation turned on (macro, primitives, extended instructions). Sending and receiving 1000 messages take total 171037 instructions. Therefore the number of instruction for passing (send and receive) one message is 171037/1000 = 171 instructions/message.

8.12 Timer

To facilitate a real-time system, some operating system functions needed to be supported. In our system, the real-time clock is the clock of running the processor. The function

```
(gettime)
```

returns the real-time clock. The function

```
(timer t)
```

sets a timer to be time-out at t cycles in the future, not earlier than (gettime)+t. A timer is used to schedule a task according to some real-time deadline.

How a timer is implemented?

A timer stores its time value as a field in PD. A timer list keeps track of the processes that have been scheduled to time-out in the future by "timer". The time value in PD is an absolute time. When a timer is set to t, the time value in PD is set to (gettime)+t. The process that executes "timer" is blocked. It is removed from the process queue and it is added to the timer list. The timer list is sorted according to the time values from earliest time to the latest. This list will be processed by a timer process which is scheduled by the supervisor, Noss.

Timer process

The time value in the list is compared to the master time (the global variable "`clock`" in the processor simulator). If it is less than the master time the owner process of this timer is awaken. As the timer list is sorted in ascending order of time value, only the first one is consulted if it is time-out then the next one is consulted and so on.

Granularity of timer

How precise the timer is depends on how often the timer process is scheduled to run. The overhead depends on this rate. It is reasonable to have the granularity at most the same as "quanta", the time interval of the interrupt. Then, the timer process can be scheduled to run after the task switcher.

What Noss needs to do?

Noss needs to run "timer" after "switchp". The time-out timer process will be queued either at the front of the process queue or the back depends on the scheduling policy. To simulate the real-time, if the timer list is not empty and the process queue is empty, then the first process in the timer list should be scheduled to be run. The master time should be updated to advance to the time value of that process. This is similar to an ordinary event-driven simulation based on time.

8.13 Lab session

We run two processes sharing two variables through a monitor. The monitor has two functions: 1) increment the value, and 2) getting the value. The first process increments the value and waits for the second process to get the value. This procedure is repeated 20 times. To synchronise both processes so that incrementing/getting value will be "in sync", another variable, "empty" is used to signal whether the value has been used. The monitor protects these global variables. The program for the experiment is shown below.

```
(let ff empty)                  ; shared variables
(let sem1)                      ; semaphore

(def mon1 (cmd) ()
   (do
   (wait sem1)
   (if (= cmd 1)                ; cmd = 1, inc ff
      (if (= empty 0)
         (do
         (set ff (+ ff 1))
         (set empty 1)))
      ; else                    ; cmd = 2, clear empty
      (if (= empty 1)
         (set empty 0)))
      ff                        ; return ff
   (signal sem1)))

(def inc () (i n)
   (do
   (set i 0)
   (while (< i 20)              ; repeat 20 times
      (do
      (set n (mon1 1))          ; inc share variable
      (set i (+ i 1))))))

(def pff () (n)
   (do
   (set n (mon1 2))             ; get share variable
   (print n)
   (while (< n 20)              ; repeat 20 times
```

```
        (do
        (set n (mon1 2))            ; get share variable
        (print n)))))               ; and print it

(def main () (p1 p2)
   (do
   (di)
   (set activep 0)                  ; clear task-list
   (set sseg 4000)                  ; allocate SS
   (set pid 1)
   (set psw (run (switchp)))        ; create switchp
   (set activep 0)                  ; clear task-list
   (set ff 0)
   (set empty 0)
   (set sem1 (initsem 1))
   (set p1 (run (inc)))
   (set p2 (run (pff)))
   (bootnos)))
```

Append the above program to Nos, name it "nos1.txt". Compile it and generate an executable code, let is be "ns1.obj". Then run it under Noss.

```
c:\test> nvm nut.obj < nos1.txt > nos1.obj
c:\test> nvm gen2.obj < nos1.obj > ns1.obj
c:\test>noss ns1.obj
load program, last address 503
DP 1008
 *  *  *  * 1 *  * 2 *  * 3 *  * 4 *  * 5 *  * 6 *
* 7 *  * 8 *  * 9 *  * 10 * * 11 *  * 12 *  * 13 *
* 14 *  * 15 *  * 16 *  * 17 *  * 18 *  * 19 *  *
20

6592 inst. 33289 cycles (system 12009 user 21280)
switchp 42

c:\test>
```

The "*" shows the task switch to run user processes. The output shows that two processes synchronised properly. There are 42 task switching, 40 comes from

switching between two processes, each 20 times.[1] The system cycle reports the number of cycle used in the Nos itself. The user cycle reports the number of cycle used to actually running the user process, (inc) and (pff). You can observe that the system consumes about one-third of the cycles.

8.14 Summary

In this chapter we have developed an operating system, Nos. The operating system is preemptive. It supports multi-thread. Two simple interprocess-communication methods have been implemented: semaphore and message passing. Some facilities for real-time processes are outlined.

It is not a surprise about the fact that Nut language can be extended minimally to write the whole operating system. The design of the extension is critical. Using the model of interrupt is a good framework to implement a simulator to run the operating system. The supervisor program, Nos supervisor (Noss), mediates between Nos and the processor simulator, Sx. The processor runs its user program continuously until an interrupt event occurs, then it hands back the control to the supervisor. The supervisor, Noss, performs the task of saving/restoring the computation state of the process to/from the process descriptor. Noss does a minimal job of intervention. Majority of the task switching and other operating system service functions are done in the user-space by Nos.

Semaphore, monitor and messaging are progressive development toward a higher abstraction which is easier to use. The behaviour of these services can be observed. The implementation is short and simple enough to be experimented with. The processor simulator gives us the detail at the level of cycle-by-cycle execution such that the effect of the system as a whole can be studied.

8.15 Further reading

Operating systems have been developed over the past 50 years. The major breakthroughs in operating system technology from the 1950s to 1990s have been

[1] What does two other task switching come from? This question is left to be investigated by the interested reader

collected in the book by Hansen [HAN01]. The earliest time-sharing systems were the Compatible Time-Sharing System (CTSS) developed at MIT [COR62] and The Multiplexed Information and Computing Services (MULTICS) [COR65]. Many textbooks cover operating systems, including Stallings [STA00], Tanenbaum [TAN01], and Silberschartz et al [SIL03].

References

[BUR01] Burns, A. and Wellings, A., Real-time systems and programming languages, 3rd ed. Addison-Wesley, 2001.

[COR62] Corbato, F., Merwin-Daggett, M., and Daley, R., "An experimental time-sharing system", Proc. of the AFIPS Fall Joint Conference, pp.335-344, 1962.

[COR65] Corbato, F., and Vyssotosky, V., "Introduction and overview of the MULTICS system", Proc. of the AFIPS Fall Joint Computer Conference, pp.185-196, 1965.

[DIJ65] Dijkstra, E., "Solution of a problem in concurrent programming control", Communication of the ACM, 8(9):569, 1965.

[HAN01] Hansen, P. (ed.), Classic Operating Systems, Springer-Verlag, 2001.

[HOA74] Hoare, C.A.R., "Monitors : an operating system structuring concept", Comm. ACM, 17(10):549-557, 1974.

[LEH89] Lehoczky, J.P., Sha, L. and Ding, Y., "The rate monotonic scheduling algorithm – Exact characterization and average case behavior", Proc. IEEE Real-time Systems Symp., pp. 166-171, 1989.

[LEI80] Leinbaugh, D.W., "Guaranteed response time in a hard real-time environment", IEEE trans. on software engineering, January 1980.

[SHA88] Sha, L., An overview of real-time scheduling algorithms, Software Engineering Institute, Carnegie Mellon University, 1988.

[SIL03] Silberschatz, A., Galvin, P., Gagne, G., Operating System Concepts, 6th ed. John Wiley, 2003.

[STA00] Stallings, W., Operating Systems, 4th ed. Prentice Hall, 2000.

[TAN01] Tanenbaum, A., Modern Operating Systems, Prentice Hall, 2001.

Exercises

8.1 Vary the quanta and observe the behaviour of Nos running some applications.

8.2 Write timer and its associated function, gettime.

8.3 Implement producer/consumer processes with a buffer of size *n*.

8.4 How to create and destroy a process dynamically?

8.5 How to improve the performance of Nos?

8.6 Nos has no input, to allow concurrency, the input can be simulated through the console application. A console accepts input and feed it as a stream to the receiving process. Write a console application.

8.7 Discuss the co-operative process. How can it be implemented? Co-operative process can be implemented at a lower cost than the pre-emptive OS. Write co-operative process in Nut and discuss its cost.

8.8 Nos has a single address space. To protect resources used by a process, a virtual memory is necessary. Discuss how to implement a virtual memory under our framework.

8.9 To implement interrupt properly, we rely on Noss as a privilege process. Noss is written in C and works in cooperation with the Sx processor simulator. Is it possible to write Noss in Nut as a user program?