OS-ACS: the kernel

Version 1.12

P.J. Maker and I.Stead

Released: 88/03/01
Printed: 88/03/01

Abstract

The kernel of the OS-ACS system provides facilities for:

1. Process scheduling.
2. Context saving.
3. Inter-process communication.

Contents

1 Introduction

1.1 Scope

This document provides the detailed design description for the kernel. The following information is included:

1. A definition of the requirements for the kernel.
2. An overview of the system for the user.
3. A description of the system generation procedure.
4. Detailed interface semantics.
5. An implementation overview.
6. A formal specification of the scheduler.
7. A summary of testing requirements.
8. A tutorial giving examples of system generation and message implementation.

1.2 Reference documents

1. 65C124 Microcomputer Detail Specification, Revision 4.
2. Amsterdam Compiler Kit Reference Manual, Chapter 2: ACK C Reference Guide.
3. ACS: the architecture, version 1.3, sw200.

1.3 Overview

The kernel provides the following facilities:

1. Process scheduling: scheduling the μP between a number of competing processes.
2. Context saving: a simple form of context saving for the processes so that they can suspend and resume execution.
3. Inter-process communication using mailboxes and pipes.

2 Requirements

2.1 Facilities

The kernel must provide the following facilities:

1. Separate processes:

   The μP will be shared between a number of distinct processes.

2. Process scheduling:

   The scheduling scheme for sharing the μP must provide:

   (a) Rapid preemption of a process when a higher priority process becomes ready to run (if the currently running process can accept it).

   (b) A method for time slicing the μP between a number of processes with the same priority.

3. Preemption protection:

   Processes must be able to prevent their preemption by other processes. For example, a process which shares a data structure with another process must be able to prevent its preemption by the partner process during the accessing of the shared data.

4. Interprocess communication:

   Facilities must be provided to enable processes to communicate. Message passing will be the principle means of communications, using mailboxes and pipes.

2.2 Constraints

The system must also satisfy the following requirements:

1. Timing requirements:

   The hardware architecture for the ACS imposes the following timing requirements:

   (a) The maximum time for which interrupts can be disabled by the fundamental operations of this system shall be less than 2ms.[1]

   (b) The time cost of rescheduling processes should be small when compared to the work done by a typical process.

[1] This is caused by the need to handshake the X92 at a high speed (lest the μP be reset).

As an aiming post let us suggest approximately 1ms per scheduling operation if there is a process ready and eligible under the scheduling rules. If there are no processes eligible, then the time should be smaller (approx. 0.3ms).

2. Maximum response times:

   For a given mix of processes and interrupt arrival rates it must be possible to calculate the maximum response time for any operation.

3. Bounded stack usage:

   The stack usage of the system must be shown to be bounded by a small constant.[2]

4. Place the μP into its low power mode:

   The μP should be put into its low power mode when no other processes are either running or ready to run, to conserve battery power.

3 User overview

This section provides an overview of the kernel for its users. It gives a description of the environment and its use without providing the detailed semantics of each component (for that information see the interface semantics section).

3.1 Definitions

The following definitions are used in this document:

process: a software object that may be scheduled to run on the μP.

class: one of ten priority divisions into which processes are categorised.

scheduling class: the class allocated to a process at system generation which determines when the process should be run.

effective class: the class that determines when a process can be preempted.

active: the state of a process when it is currently running on the μP.

preempted: a process is preempted if it has been halted by the scheduler to allow a higher priority process to run.

[2] The 65C124 has a stack of 192 bytes. Since two of these locations are used for temporaries our stack must be smaller than 190 bytes.

ready: a process is ready if it is waiting for the scheduler to make it active.

dormant: the state of a process when it is not active, preempted or ready to run.

context: the information which must be saved when a process is suspended by the scheduler. For example the program counter.

messages: messages provide the principle means of communication between different processes in this system. For example, process 1 may send a message to process 2 indicating that a tachyarrhythmia has been detected. Messages may be passed using mailboxes or pipes.

mailbox: a data structure which may contain a single piece of information.

pipe: a byte stream of fixed length.

sender: a process that constructs and sends messages to a particular mailbox or pipe. There may be many senders to one mailbox or pipe.

server: a process that receives messages from a particular mailbox or pipe. Only one server can exist for each mailbox or pipe, and servers can be changed at run time.

3.2 Processes

Processes are implemented as C functions which are called from within the scheduling part of the kernel. These processes have the following properties:

1. All processes are declared when the system is built. It is not possible to create or destroy processes at run time in this system.

2. There can be at most 255 different processes in this system. Processes are identified by constants of type `PROCESS_ID`. The 256th identifies a non-existent process `NO_SUCH_PROCESS`.

3. Any process or function can make another process in the system ready to run by calling the make_process_ready function using a system wide set of process identifiers. Once a process is ready to run then it will obey the scheduling rules described in section 3.3.

   Note that any number of make_process_ready operations which occur before a process is running have the same effect as one operation and result in only one activation of the process.

4. A process which is ready to run will eventually be called by the scheduling system[3]. Note that a process can, of course, make itself ready to run. If a process is made ready while it is already running, it will eventually be restarted in accordance with the scheduling rules after it returns to the scheduling system.

3.3 Process scheduling

Processes are divided into ten different priority classes, which enables scheduling to be handled in two phases:

1. Scheduling the μP between the different classes of processes.

   This part of the scheduler selects which class of processes should be run on the μP.

2. Scheduling the processes within a single class.

   Each class has its own scheduler which selects between the processes of that class.

3.3.1 Scheduling the μP between the different classes of processes

This section describes how the system selects which class of processes to run.

3.3.1.1 PROCESS CLASSES The processes which are scheduled by the kernel are divided into ten different classes. These are labelled 8, 7, .., 0, -1. They correspond to:

class 8: hardware interrupt handlers and other non-preemptible processes.

classes 7 to 0: eight classes of "normal" processes.

class -1: this is a special class containing one process which runs on the system whenever no other process is ready to run. The class -1 process could simply place the μP into its low power mode or be used for measuring the duty cycle of the system.

3.3.1.2 PRIORITY BETWEEN CLASSES OF PROCESSES A higher class number indicates a higher priority within this scheduling system. Thus, processes of class n can be preempted by processes of class m if $m > n$.

[3]Provided that the other processes allow it to run.

For example, a class 4 process can only be preempted by processes from classes 5,6,7 and 8. Class 8 processes can never be preempted, since there are no higher priority processes.

Note that class n processes cannot be preempted by other class n processes without their active request.[4]

At any time there can be no more than one member of each class which is either active or preempted; all other processes in that class must be either dormant or ready.[5]

3.3.1.3 PREVENTING PREEMPTION A process needs to be able to prevent its preemption by higher priority processes at certain times. This may be for the following reasons:

1. To ensure mutual exclusion of access to data.
2. Because of timing constraints.

We achieve this by allowing a process to temporarily promote itself into a higher priority class. This promotion is achieved by using the set_effective_class function.[6]

For example a class 2 process could use set_effective_class(4) which would prevent all preemption by processes from classes 3 and 4 (as well as the usual −1, 0, 1 and 2). At some later time it would reset its effective class back to its scheduling class.

It is, however, illegal for a process to set its effective class to a lower level than its scheduling class. If such an operation was legal then the preemption protection system could be broken.

Consider the following example:

```
process P which is in class 2
{
        old_ec = set_effective_class(4) ;        /* prevent preemption by class 3 & 4 */
        <P Work....> ;
        old_ec = set_effective_class(2) ;        /* restore class level to default */
```

[4] For example two class n processes may wish to share the μP amicably between themselves. This requires an ability for a process to ask for its own preemption (and eventual resumption).

[5] Since there can only be one process from each class in active or preempted then the stack has a finite size ($\Sigma_{class=-1}^{8} max(stack\_usage(class))$).

[6] The set_effective_class function changes the effective class a process belongs to. These ideas are based on the PDP-11 notion of processor levels for interrupt handling. Refer to the SPL instruction in the PDP-11/45 and PDP-11/70 architectures.

```
}

process Q which is in class 5
{                                                                    10
        /* currently in class 5 */
        old_ec = set_effective_class(4) ;        /* let a class 4 process run for me */
        <Q Work....> ;
        old_ec = set_effective_class(5) ;        /* restore it back to normal */
}
```

Then if Q interrupts P then P can be preempted by a class 4 process while it is performing <P Work...>. This contradicts the purpose of the set_effective_class(4) in P.

3.3.2 Scheduling processes within a class

Since processes within a class cannot preempt each other, scheduling within a class is simple.

The scheduler for a class simply continues selecting and running ready jobs within its own class until there are no more ready to run. Once a process has been started, it continues to run until completion or until it makes an explicit request to be halted.

Two alternative scheduling schemes are provided for scheduling within classes. One scheme selects processes on a priority basis, whilst the other selects them in a cyclical sequence.

3.4 Context saving

Scheduling within a class occurs when the current process returns to the scheduler that called it. If a process wishes to return to the scheduling system it may need to save some of its context.

This system currently provides a very limited form of context saving, in which only the program counter is saved and restored. A process can only return to its class scheduler when there is only a program counter on the system stack. So for a process with this form of context saving we require:

1. No local (stack-based) variables.
2. No return program counters; i.e. you cannot suspend inside a function called from your outermost process.

3.5 Communication

The processes need to communicate in order to perform their functions. Two methods of message passing are defined for this system:

1. Mailboxes:

   A mailbox is a data structure containing a single piece of user defined information.

2. Pipes:

   A pipe is a byte stream of fixed length. A pipe is used for communication when a mailbox would not provide sufficient capacity.

   A valuable feature of fixed length pipes is that even if the head and tail of a pipe are corrupted, store outside the pipe will not be overwritten.

   NOTE. Dynamic assignment of servers to pipes is not possible at the moment.

4 System generation

The kernel directory contains a number of options that may be selected by the user. By defining the required options, the user builds a library specific to the particular set of processes and scheduling constraints in the user's system. Default values are provided for any options not explicitly defined by the user.

In this section, the system generation procedure is described for a general system called X.

Examples of system generation are provided in section 8.

4.1 Input

The input to the system generation procedure is the file X.sys - the description of the system. The X.sys file should contain the following information:

1. `/* ... */`

   C style comments are supported and may be included as required.

2. `BEGIN`

   This keyword introduces the start of the table. It exists simply for the assistance of the implementing entity.

3. SCHEDULER USES LINEAR SEARCH

   This is an option which makes the scheduler use linear search when selecting which class of processes to run. By default the scheduler uses binary search.

4. KERNEL TRACE USING PRINT

   This is an option which enables the kernel to produce trace messages using printf.

5. SCHEDULER -1 IS *Y*

   The class -1 scheduler is defined by this statement. Since we normally have only one class -1 process, which never returns, we can define the scheduler to be that process.

   The kernel provides the following possiblities for the value *Y*.

   (a) `lowpower`

   This process simply puts the μP into its low power mode. (Necessary for using this system in a real implant). This is the default value.

   (b) `dutycycle`

   A process which measures the dutycycle of the μP.

   Of course the user could define it to be another process if required.

6. SCHEDULER *CLASS* IS *TYPE*

   This statement defines the TYPE of scheduler for each process CLASS.

   CLASS can take any value in the range 0,1..7.

   TYPE should be one of the following:

   (a) ROUNDROBIN

   This scheduler runs the ready processes in turn, starting with the process that has the smallest process identifier. This is the default value.

   (b) PRIORITY

   Each time this scheduler is called, it selects between the ready processes in this class on a priority basis.

   (c) SPECIAL

   The user can define his own scheduler if required.

7. CLASS *N* PROCESS *PID* AS *func* CALLED *"name"*

   To declare a process in the system the following information must be provided:

(a) *N* - the class to which the process belongs (0,1..7).

(b) *PID* - the name of this process (for system calls).

(c) *func* - the function that implements this process.

(d) *"name"* - a string which is used as the debugging name for this process.

Processes of different classes may be declared in any order, but processes within the same class are allocated priority in the order in which they are declared.

8. *INT* `HANDLER IS` *proc*

   The handler for interrupt source *INT* is *proc*. If the user does not define a handler for some interrupt, the default value is for the interrupt handler to call the `illegal_interrupt` function.

9. `END`

   This announces the end of the X.sys file.

4.2 Output

The outputs of system generation are:

1. X.a - the ACK(1) library containing the version of the kernel specific to the system defined in X.sys.
2. X.h - the include file describing the functions and processes defined in X.sys.
3. X.fil - the filter program which checks that the scheduling rules of the kernel are being obeyed, expands the kernel's brief trace messages into text messages and prints out the processes on the stack whenever the kernel's state changes.

4.3 Generating the system

To generate the system in the work directory, enter the following command:

```
sysgen work X.sys
```

work could also be `frozen` or `released` as required.

5 Interface semantics

This section describes the interface and its associated semantics.

The description of each function includes:

1. A description of the function, including its arguments and types.
2. A formal description of the effect of these functions.

5.1 Process scheduling interface

The interface to the process scheduling system provides facilities for:

1. Making processes ready to run.
2. Changing the effective class membership of this process.

5.1.1 void make_process_ready(PROCESS_ID p)

This function will make the process identified by p ready to run. In particular it will:

1. Update the kernel data structures to indicate that this process is ready to run.
2. Start this process if required by the scheduling priority rules.

PRECONDITIONS System intact.[7] VALID_PROCESS_ID(p)

INCONDITIONS Process p will become active and run to completion if $sc(p) > ec(active)$[8]. Interrupts disabled.

POSTCONDITIONS Process p will be ready if $sc(p) \leq ec(active)$.

5.1.2 EFFECTIVE_CLASS set_effective_class(EFFECTIVE_CLASS nec)

This function will change the effective class of the currently running process to *nec*. *nec* should be greater than or equal to the scheduling class of the process.

EFFECTIVE_CLASS is in the range EC_0, EC_1..EC_8.

[7] The data structures in this system are in order.

[8] sc = scheduling class, ec = effective class

PRECONDITIONS System intact.

INCONDITIONS Processes $Q_0..Q_m$ will be running if $sc(Q_n) > ec(active)$ Interrupts disabled while effective class is being changed (but not while other processes run).

POSTCONDITIONS The effective class of this process is set to *nec* and its old effective class is returned.

ASSERTIONS $nec \in \{EC\_M1, ...EC\_8\}$

5.1.3 inline void BEGIN_CRITICAL(), END_CRITICAL()

These are used to delimit a critical region of code. All interrupts will be disabled between these two calls. Unlike the setting of effective class this can be done very quickly (and should probably be used for any operation which requires less than $\frac{1}{2}$ms).

The old value of the interrupt flag is saved on the μP stack so it is illegal to return from a function which has called BEGIN_CRITICAL without executing the corresponding END_CRITICAL.

INCONDITIONS BEGIN_CRITICAL() uses 5 clock cycles, END_CRITICAL() uses 4 clock cycles. For every call to BEGIN_CRITICAL there must be a corresponding call to END_CRITICAL within the same function. No procedure returns and no calls to kernel services can occur between the corresponding calls to BEGIN_CRITICAL and END_CRITICAL.

POSTCONDITIONS If interrupts were enabled before BEGIN_CRITICAL, then they are enabled, otherwise interrupts are disabled.

5.2 Mailboxes

5.2.1 MAILBOX(<declarations>)

This is used when constructing the system to define the type of the mailbox.

<declarations> is a list of the data in the mailbox type. For example, the type of a mailbox that contains events and the intervals between them might be defined as:

typedef MAILBOX(EVENT_TYPE event ; SHORTTIME interval ;) MOM ;

5.2.2 INITIAL_MAILBOX(SERVER)

This defines the initial values for the mailbox whose server is $SERVER$. If required, $SERVER$ could be defined as $NO\_SUCH\_PROCESS$.

For example:

MOM my_mom = INITIAL_MAILBOX(SERVER) ;

5.2.3 RESET_MAILBOX(M,S)

This is used if it is necessary to reset a mailbox to its empty state. It makes the mailbox M empty and sets its server to be S.

5.2.4 lvalue ACCESS_MAILBOX(MBOX,FIELD)

This returns an lvalue[9] for the user defined $FIELD$ in the mailbox $MBOX$.

For example:

ACCESS_MAILBOX(m,interval) = 10 ;

5.2.5 PROCESS_ID GET_SERVER(M)

This returns the server process for mailbox M.

5.2.6 bool FULL_MAILBOX(M)

This returns true if mailbox M is full (not empty).

5.2.7 bool EMPTY_MAILBOX(M)

This returns true if mailbox M is not full (empty).

5.2.8 void SET_SERVER(M,S)

This is used to set the server process for mailbox M to be process S.

[9] An lvalue can occur on both sides of the assignment operator. See the C manual for the definition of lvalue.

PRECONDITIONS $S \in PROCESS\_ID$

POSTCONDITION $SERVER(M) = S$

5.2.9 FREE_SERVER(M)

PRECONDITION The server for mailbox M must be the caller process of $FREE\_SERVER$.

POSTCONDITION $SERVER(M) = NO\_SUCH\_PROCESS$

5.2.10 void print_mailbox(&MAILBOX);

This prints the contents of $MAILBOX$ via the UART.

5.2.11 Sending to a mailbox

```
SEND_MAILBOX(m){
        <ACCESS the fields>
}ON_MAILBOX_OVERFLOW(m){
        <Overflow handler>
}END_SEND_MAILBOX() ;
```

PRECONDITIONS System intact.

INCONDITIONS Interrupts disabled. If the mailbox is empty, we access the fields and make the server ready. The time to access the fields must be <2ms.

POSTCONDITIONS If the mailbox is not empty, the overflow handler is called.

5.2.12 Reading a mailbox

```
if( FULL_MAILBOX(M)) {
        READ_MAILBOX(M){
                <accessing code>
        }END_READ_MAILBOX(M) ;
}
```

5.3 Pipe construction

5.3.1 type PIPE

This declares the type $PIPE$.

5.3.2 INITIAL_PIPE(SERVER)

This is used to initialise a pipe.

For example,

PIPE P INITIAL_PIPE(PROC 1)

defines $PROC1$ to be the server for pipe P.

5.3.3 void RESET_PIPE(PIPE,SERVER)

This resets pipe $PIPE$ by setting the head and tail to 0 and the server to $SERVER$.

5.4 Pipe primitives

These are low level operations on pipes.

Process preemption must be disabled around sets of operations.

5.4.1 byte READ_PIPE(P)

This reads the first byte from the head of pipe P.

PRECONDITIONS P is not empty.

POSTCONDITIONS $return = head(p.data) \wedge p.data' \doteq tail(p.data)$

5.4.2 inline void WRITE_PIPE(p,d)

This adds data d onto the end of pipe p provided that:

$p \in PIPE$

$d \in byte$

PRECONDITIONS p is intact.

POSTCONDITIONS $\#(p.data) < 255 \Rightarrow p.data' = p.data^\frown[d]$

$\#(p.data) = 255 \Rightarrow pipe\_overflow(\&p)$ *is called*

ASSERTIONS p is not empty.

5.4.3 bool EMPTY_PIPE(P)

PRECONDITIONS $P \in PIPE$ and P is intact

POSTCONDITIONS If P.data = []

5.4.4 void READY_PIPE_SERVER(P)

PRECONDITIONS System intact.

POSTCONDITIONS Makes the server process for pipe P ready.

5.4.5 byte PIPE_USED(P)

This returns the number of bytes used in pipe P.

5.5 High level pipe operations

5.5.1 void write_pipe(&pipe,&info,nbytes)

This will write *nbytes* from address $info$ to *pipe*, and then make the server process ready.

PRECONDITIONS $\#(pipe.data) + nbytes \leq 255$

POSTCONDITIONS $pipe.data' = pipe.data^\frown info$

make_process_ready(pipe.server)

5.5.2 byte read_pipe(&pipe,&info,nbytes)

This will read between 0 and *nbytes* of data from *pipe* into address $info$.

PRECONDITIONS True.

POSTCONDITIONS return value = # bytes read = min(nbytes,# pipe.data)
info[0,,nbytes-1] = bytes read
pipe.data = tail(min(nbytes,# pipe.data),pipe.data)

6 A formal specification of the scheduler

This section provides a formal specification of the process scheduling system using Z.

6.1 Definitions

The processes in the system are represented with:

$$Process \triangleq 0..Max_{process} - 1$$

Processes are divided into classes for the purposes of scheduling and preemption. These are of course defined by:

$$Class \triangleq -1..8$$

The μP is represented by a single process which is running (*active*) together with the set of processes which have been interrupted and have their activation records on the μP stack (*preempted*).

Machine

$$active : Process$$
$$preempted : \mathbf{P}\ Process$$

$$active \notin preempted$$

We also define the traditional Z schemas for changing and maintaining state.

$$\Delta Machine$$
$$Machine$$
$$Machine'$$

$$\Xi Machine$$
$$\Delta Machine$$

$$active' = active$$
$$preempted' = preempted$$

We also need a representation for the processes which are ready to run in this system. The *ready* set will never be empty (since a process exists that keeps itself ready forever). This is given by:

$$Ready$$
$$ready : \mathbf{P}\ Process$$

$$ready \neq \{\}$$

And ...:

$$\Delta Ready$$
$$Ready$$
$$Ready'$$

$$\Xi Ready$$
$$\Delta Ready$$

$$ready = ready'$$

The class membership of processes is represented by the schema *Classes*. The *sc* function defines the scheduling class for a process (i.e. its priority). The *ec* function defines the effective class of a process for the purposes of avoiding preemption.

Classes

$$sc : Process \longrightarrow Class$$
$$ec : Process \longrightarrow Class$$

$$sc' = sc$$
$$\forall p : Process \cdot ec(p) >= sc(p)$$

And ...:

$\Delta Classes$

$$Classes$$
$$Classes'$$

$\Xi Classes$

$$\Delta Other$$

$$sc = sc'$$
$$ec = ec'$$

6.2 Making processes ready

How do we make a process ready to run (and possibly start it)? The schema MR defines this operation. It simply adds the input process $p?$ into the set of *ready* processes and then does a *Schedules* operation.

$MR \triangleq MR_0 ; Schedule$

MR_0

$$\Xi Machine$$
$$\Xi Classes$$
$$\Delta Ready$$
$$p? : Process$$

$$ready' = ready \cup \{p?\}$$

How do we define the *Schedule* operation? Well, we let p be the process we could possibly run from the sets *ready* and *preempted*. This is defined by the

function *pick*. Then we make it the *active* process and readjust the *preempted* set if necessary.

Schedule

$$\Xi Classes$$
$$\Delta Ready$$
$$\Delta Machine$$
$$p : Process$$

$$p = pick(ready \cup preempted)$$
if $sc(p) > ec(active)$ **then**
$$ready' = ready - p$$
$$active' = p$$
$$preempted' = preempted \cup \{active\} - p$$
else
$$\Xi Machine$$
$$\Xi Ready$$

The *pick* function selects a member from the highest priority set defined by the function *hp*. This selection function is left unspecified at present. It is really defined by the scheduling schemes which are implemented within a single class.

$$pick : \mathbf{P}\ Process \longrightarrow Process$$

$$pick(S) = x \textbf{ where } x \in hp(S)$$

]

What is *hp*? It's the set of highest priority processes within its argument. That is, for some priority function *priority* we have:

$$hp : \mathbf{P}\ Process \longrightarrow \mathbf{P}\ Process$$

$$hp(S) = \{x | x \in s \wedge (\forall y \in S \cdot priority(x) >= priority(y))\}$$

And what is the priority? It is the scheduling class plus a small loading for being preempted.

$$priority : Process \longrightarrow \mathbf{N}$$

$$\textbf{if } p \in preempted \textbf{ then } 2 * ec(p) + 1 \textbf{ else } 2 * sc(p)$$

6.3 Set Effective Class

To set the effective class of the currently active process we simply change the *ec* function and perform a *Schedule* operation.

$SEC \triangleq SEC_0; Schedule$

SEC_0

$\Xi Machine$
$\Xi Ready$
$\Delta Classes$
$c? : Class$

$c? >= sc(active)$
$ec' = ec \oplus \{active \mapsto c?\}$

6.4 Essential conditions

These conditions are fundamental to process scheduling.

$$\forall x, y \in preempted \cdot x \neq y \Rightarrow sc(x) \neq sc(y)$$

$$|preempted| <= 8$$

$$\forall p \in Process \cdot ec(p) >= sc(p)$$

7 Testing

The principles listed below will be adopted for testing the kernel:

1. We shall test the scheduler between classes of processes exhaustively (256 cases).
2. The schedulers for a single class shall be tested for 0, 1, 2, 3, 15, 16, 17 processes (again exhaustively).

3. A random test will be run on the system which will generate a trace of the scheduling events. This will then be checked to see if it conforms to the scheduling rules.
4. Context switching calls will be tested by example.
5. Message passing will again be tested.

8 Tutorial

This section provides a tutorial to demonstrate how to set up the kernel to suit the requirements of a particular system.

8.1 System generation

This example system, ex_P, contains the following processes:

- One class 0 process which initialises the system.
- Two class 2 processes, TEST0 and TEST1.
- A class 5 process which performs a high priority task.
- A class 7 process which provides a trace of system events.

8.1.1 The ex_P.sys file

This file contains the definition of the system:

```
BEGIN

CLASS 0 PROCESS INITIALISE AS initialise CALLED "initialise"
CLASS 7 PROCESS TRACE_SERVER AS trace_server CALLED ""

CLASS 2 PROCESS TEST0 AS test0 CALLED "test #0"
CLASS 2 PROCESS TEST1 AS test1 CALLED "test #1"

CLASS 5 PROCESS HP AS hp CALLED "high priority job"

END
```

8.1.2 The ex_P.c file

This file contains the C functions that implement the system.

```
#define         DODEFINE
#include        <stdc.h>
#include        <ex_P.h>
#include        <trace.h>
#include        <utilities.h>

#define EX_LOG0(S) LOG(0x0001L,(STR_LOGMSG(0,S)))

PROCESS initialise() /* bring up the world */
{
        addclasses(0xFFFFL) ;
        make_process_ready(TEST0) ; /* start each test in turn */
        make_process_ready(TEST1) ;
        exit(1) ;                   /* finished it */
}

PROCESS hp() /* high priority process */
{
        EX_LOG0("HP\n") ;
}

PROCESS test0() /* get preempted by HP */
{
        EX_LOG0("test #0: make_process_ready(HP) ;\n") ;

        make_process_ready( HP ) ;

        EX_LOG0("          back in test0\n") ;
}

PROCESS test1() /* set effective class example */
{
        LOCAL EFFECTIVE_CLASS old_ec ;

        EX_LOG0("test #1: raise our effective class and see that\n") ;
        EX_LOG0("    it stops us from being preempted\n")  ;

        old_ec = set_effective_class(EC_7) ;
        EX_LOG0("    (HP should NOT start until the next message\n") ;
```

```
        make_process_ready( HP ) ;
        /* Do some work safe from preemption by processes in classes -1..7 */

        EX_LOG0("      (HP must start now that we lower our effective class\n");
        (void) set_effective_class( old_ec ) ;
        EX_LOG0("      back in test1\n") ;
}
```

60

8.1.3 Trace messages

The following trace messages are produced when the ex_P system is run using filter ex_P.fil:

```
@make_process_ready(test #0@2) ; ready=(test #0@2 ) ; running=();
@start_process(test #0@2) ; ready=() ; running=(test #0@2 );
test #0: make_process_ready(HP) ;
@make_process_ready(high priority job@5) ; ready=(high priority job@5
) ; running=(test #0@2 );
@start_process(high priority job@5) ; ready=() ; running=(test #0@2
high priority job@5 );
HP
@end_process(high priority job@5) ; ready=() ; running=(test #0@2 );
         back in test0
@end_process(test #0@2) ; ready=() ; running=();
@make_process_ready(test #1@2) ; ready=(test #1@2 ) ; running=();
@start_process(test #1@2) ; ready=() ; running=(test #1@2 );
test #1: raise our effective class and see that
    it stops us from being preempted
@set effective class 7
    (HP should NOT start until the next message
@make_process_ready(high priority job@5) ; ready=(high priority job@5
) ; running=(test #1@2 );
     (HP must start now that we lower our effective class
@set effective class 2
@start_process(high priority job@5) ; ready=() ; running=(test #1@2
high priority job@5 );
HP
@end_process(high priority job@5) ; ready=() ; running=(test #1@2 );
     back in test1
@end_process(test #1@2) ; ready=() ; running=();
```

8.2 Setting up mailboxes

This example demonstrates the procedure for setting up mailboxes.

8.2.1 The ex_M.sys file

This file contains the definition of the system in which messages are to be used.

```
BEGIN

CLASS 0 PROCESS INITIALISE AS initialise CALLED "initialise"
CLASS 7 PROCESS TRACE_SERVER AS trace_server CALLED ""

CLASS 2 PROCESS TEST0 AS test0 CALLED "test #0"
CLASS 3 PROCESS LISTENER AS listener CALLED "listener"

END
```

8.2.2 The ex_M.c file

This file contains the C functions that implement the messages.

```
#define         DODEFINE
#include        <stdc.h>
#include        <ex_M.h>
#include        <trace.h>
#include        <utilities.h>

#define EX_LOG0(S) LOG(0x0001L,(STR_LOGMSG(0,S)))
#define EX_LOG1(S,D) LOG(0x0001L,(STR_LOGMSG(2,S),D))
#define EX_LOG2(S,D,E) LOG(0x0001L,(STR_LOGMSG(4,S),D,E))

PROCESS initialise()
{
        addclasses(0xFFFFL) ;
        make_process_ready(TEST0) ;
        puts("About to die\n") ;
        exit( 1 ) ;
}

LOCAL MAILBOX( byte event ; byte number ) mbox0
                                = INITIAL_MAILBOX(LISTENER) ;
```

```
LOCAL MAILBOX( byte job ) mbox1 = INITIAL_MAILBOX(LISTENER) ;

PROCESS listener()
{
        for(;;) {
                if( FULL_MAILBOX(mbox0)) {
                        READ_MAILBOX(mbox0) {                                   30
                                EX_LOG2("listener: mbox0 is %d %d\n",
                                        ACCESS_MAILBOX(mbox0,event),
                                        ACCESS_MAILBOX(mbox0,number)
                                ) ;
                        } END_READ_MAILBOX(mbox0) ;
                } else if( FULL_MAILBOX(mbox1)) {
                        READ_MAILBOX(mbox1) {
                                EX_LOG1("listener: mbox1 is %d\n",
                                        ACCESS_MAILBOX(mbox1,job)
                                ) ;                                             40
                        } END_READ_MAILBOX(mbox1) ;
                } else {
                        EX_LOG0("listener: no messages\n") ;
                        return ;
                }
        }
}

PROCESS test0() /* send job numbers 0..10 to mbox1 */
{                                                                               50
        byte i ;

        for( i = 0 ; i != 11 ; i++ ) {
                SEND_MAILBOX(mbox1) {
                        ACCESS_MAILBOX(mbox1,job) = i ;
                } ON_MAILBOX_OVERFLOW(mbox1) {
                        EX_LOG0("overflow ????\n") ;
                } END_SEND_MAILBOX(mbox1) ;
        }
}                                                                               60
```

8.2.3 Trace messages

The following trace messages are produced when the ex_M system is run using filter ex_M.fil:

```
@make_process_ready(test #0@2) ; ready=(test #0@2 ) ; running=();
@start_process(test #0@2) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 0
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 1
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 2
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 3
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 4
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 5
```

```
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 6
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 7
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 8
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 9
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
listener: mbox1 is 10
listener: no messages
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@end_process(test #0@2) ; ready=() ; running=();
About to die
```

8.3 Setting up pipes

This example demonstrates the procedure for setting up pipes.

8.3.1 The ex_PP.sys file

This file contains the definition of the system in which pipes are to be used.

```
BEGIN

CLASS 0 PROCESS INITIALISE AS initialise CALLED "initialise"
CLASS 7 PROCESS TRACE_SERVER AS trace_server CALLED ""

CLASS 2 PROCESS TEST0 AS test0 CALLED "test #0"
CLASS 3 PROCESS LISTENER AS listener CALLED "listener"

END
```

8.3.2 The ex_PP.c file

This file contains the C functions that implement the pipes.

```
#define         DODEFINE
#include        <stdc.h>
#include        <ex_PP.h>

#define EX_LOG0(S)  LOG(0x0001L,(STR_LOGMSG(0,S)))
#define EX_LOG1(S,D)  LOG(0x0001L,(STR_LOGMSG(2,S),D))
#define EX_LOG2(S,D,E)  LOG(0x0001L,(STR_LOGMSG(4,S),D,E))

PROCESS initialise()
{
        addclasses(0xFFFFL) ;
        make_process_ready(TEST0) ;
        exit( 1 ) ;
}

LOCAL PIPE pipe = INITIAL_PIPE(LISTENER) ;

PROCESS listener()
{
        LOCAL byte n ;
```

```
        LOCAL byte buffer[80] ;

        while((n = read_pipe(&pipe,&buffer[0],sizeof(buffer))) != 0 ) {
                byte i ;
                for( i = 0 ; i != n ; i++ ) {
                        printf("%d\n", buffer[i] ) ;
                }
        }
}

PROCESS test0()
{
        LOCAL byte buffer[90] ;

        int i ;
        for( i = 0 ; i != sizeof(buffer) ; i++ ) {
                buffer[i] = i ;
        }
        write_pipe(&pipe,&buffer[0],sizeof(buffer)) ;
}
```

8.3.3 Trace messages

The following trace messages are produced when the ex_PP system is run using filter ex_PP.fil:

```
@make_process_ready(test #0@2) ; ready=(test #0@2 ) ; running=();
@start_process(test #0@2) ; ready=() ; running=(test #0@2 );
@make_process_ready(listener@3) ; ready=(listener@3 ) ; running=(test
#0@2 );
@start_process(listener@3) ; ready=() ; running=(test #0@2 listener@3
);
0
1
2
3
4
5
6
7
8
9
```

```
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
77
78
79
80
81
82
83
84
85
86
87
88
89
@end_process(listener@3) ; ready=() ; running=(test #0@2 );
@end_process(test #0@2) ; ready=() ; running=();
```