Chapter 9. Signals and Signal Processing

Abstract: Chapter 9 covers signals and signal processing. It presents a unified treatment of signals and interrupts, which helps clarify the intended purpose of signals as a unified way to handle exceptions, rather than as a mechanism for inter-process communication. It explains signals and signal processing in Unix, and it describes the implementation of signals and signal processing in MTX in details. It shows how to install signal catchers to allow processes to handle exceptions in user mode, and it uses examples to demonstrate signal catchers for interval timer, divide exception and other simulated traps.

9.1. Signals and Interrupts

In Chapter 8, we have seen that interrupts are requests sent to a CPU, which divert the CPU from its normal executions to do interrupt processing. Like interrupts to a CPU, signals are requests sent to a process, which divert the process from its normal executions to do signal processing. Before discussing signals and signal processing, we shall review the concepts and mechanism of interrupts, which help put signals in an OS into a proper perspective.

(1). First, we generalize the notion of process to mean: a "process" (in quotes) is a sequence of activities. Examples of generalized "processes" include

. a person, who carries on his/her daily routine chores.
. a Unix (or MTX) process, which runs in its address space(s).
. a CPU, which executes machine instructions.

(2). An "interrupt" is an event delivered to a "process", which diverts the "process" from its normal activities to do something else, called "interrupt processing". The "process" may resume its normal activities when it finishes processing the “interrupt”.

(3). The term "interrupt" can be applied to any "process", not just to a CPU in a computer. For example, we may speak of the following kinds of “interrupts”.

(3).1. PERSON interrupts:
While I am reading, grading, day-dreaming, etc. in my office, some real events may occur, such as

<table>
<thead>
<tr>
<th>Real Events</th>
<th>ID</th>
<th>Action Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building on fire</td>
<td>1</td>
<td>Get out immediately!</td>
</tr>
<tr>
<td>Telephone rings</td>
<td>2</td>
<td>Pick up phone to chat with the caller.</td>
</tr>
<tr>
<td>Knock on door</td>
<td>3</td>
<td>Yell come in (or pretend not there).</td>
</tr>
<tr>
<td>Cut own finger</td>
<td>4</td>
<td>Apply band-aid.</td>
</tr>
</tbody>
</table>

-----------------------------------------------
All these may be called PERSON interrupts since they divert a person from his/her normal activities to "process or handle the interrupt". After processing an interrupt, a person may resume whatever he/she was doing before (if the person survives and still remembers what he/she was doing before). Each interrupt is assigned a unique ID number for identification, and has a pre-installed action function, which a person can "execute" upon receiving an interrupt. Depending on their origin, interrupts may be classified into 3 categories:

- From hardware: building on fire, alarm clock goes off, etc.
- From other person: phone call, knocking on door, etc.
- Self-inflicted: cut own finger, eat too much, etc.

Depending on their urgency, interrupts can be classified as

- Non-maskable (NMI): Building on fire!
- Maskable: Knocking on door, etc.

Each of the action functions of a PERSON is installed either by instincts or by experience. It is impossible to complete the above table since there are too many different kinds of PERSON interrupts, but the idea should be clear.

(3).2. PROCESS interrupts:
These are interrupts sent to a process. While a process is executing, it may receive interrupts from 3 different sources:

- From hardware: Control_C key from terminal, interval timer, etc.
- From other process: kill(pid, SIG#), death_of_child, etc.
- Self-inflicted: divide by zero, invalid address, etc.

Each process interrupt is converted to a unique ID number, which is delivered to the process. Unlike PERSON interrupts, which has too many kinds, we can always limit the number of interrupts to a process. In Unix, process interrupts are called SIGNALS, which are numbered 1 to 31. For each signal, a process has an action function in its PROC structure, which the process can execute upon receiving a signal. Similar to a person, a process may mask out certain kinds of signals to defer their processing. If needed, a process may also change its signal action functions.

(3).3. HARDWARE Interrupts:
These are signals sent to a processor or CPU. They also originate from 3 possible sources:

- From hardware: Timer, I/O devices, etc.
- From other processors: FFP, DMA, other CPUs in a multiprocessor system.
- Self-inflicted: divide by zero, protection error, INT instruction.

Each interrupt has a unique interrupt vector number. The action function is an interrupt handler in the interrupt vector table. Recall that a CPU is always executing a process. The CPU does not cause any self-inflicted interrupts (unless faulty). Such interrupts are due to the process that is using or, in most cases, misusing the CPU. The former includes the INT n or equivalent instructions, which cause the CPU to switch from Umode to Kmode.
The latter includes all trap errors recognized by the CPU as exceptions. Therefore, we may rule out the self-inflicted interrupts from a CPU, leaving only those external to the CPU.

(3).4. Trap Errors of Process:

A process may cause self-inflicted interrupts to itself. Such interrupts are due to errors, e.g. divide by 0, invalid address, illegal instruction, privilege violation, etc. which are recognized by the CPU as exceptions. When a process encounters an exception, it traps to the OS kernel, converts the trap reason to a signal number and delivers the signal to itself. If the exception occurs in user mode, the default action of a process is to terminate, with an optional memory dump for debugging. As we shall see later, a process may replace the default action function with a signal catcher, allowing it to handle signal in user mode. If the trap occurs in kernel mode, which must be due to hardware error or, most likely, bugs in the kernel code, there is nothing the kernel can do. In Unix/Linux, the kernel simply prints a PANIC error message and stops. Hopefully the problem can be traced and fixed in the next kernel release. MTX adopts the same humble attitude. Despite my best effort, I have no doubt there are still bugs in the MTX kernel.

9.2. Examples of Unix/Linux Signals

(1). Pressing the Control_C key usually causes a running process to terminate. Here is why. The Control_C key generates a keyboard hardware interrupt. The keyboard interrupt handler converts the Control_C key to a SIGINT(2) signal delivered to all processes on the terminal and wake up such processes if they are waiting for keyboard inputs. While in Kmode, every process is required to check and handle outstanding signals. For most signals, the default action of a process is to call the kernel's kexit(exitValue) function to terminate. In Linux, the low byte of exitValue is the signal number that caused the process to terminate.

(2). The user may use the nohup a.out & command to run a process in the background. The process will continue to run even after the user logout. The nohup command causes the sh to fork a child to execute the program as usual, but the child ignores the SIGHUP(1) signal. When the user logout, the sh sends a SIGHUP signal to all processes associated with the terminal. Upon receiving such a signal, the background process simply ignores it and continues to run. To prevent the background process from using the terminal for I/O, the background process usually disconnects itself from the terminal (by redirecting its file descriptors 0,1,2 to /dev/null), making it totally immune to any terminal oriented signals.

(3). Perhaps a few days later the user login again and finds (by ps -u UID) that the background process is still running. The user may use the sh command

```
kill pid (or kill -s 9 pid)
```

to kill it. Here is how. The process executing kill sends a SIGTERM(15) signal to the target process identified by pid, requesting it to die. The targeted process will comply with the request and terminate. If the process has chosen to ignore the SIGTERM signal, it may refuse to die. In that case, we may use kill -s 9 pid, which will kill it for sure. This is because processes cannot change their actions for the number 9 signal. The reader may
wonder, why number 9? In the original Unix, there were only 9 signals. The number 9 signal was reserved as the last resort to kill a process. Although later Unix systems expand the signal numbers to 31, the meaning of signal number 9 is still retained.

9.3. Signal Processing in Unix

(1). Signal types in Unix: Unix supports 31 different signals, which are defined in the signal.h file. Each signal has a symbolic name, such as SIGHUP(1), SIGINT(2), SIGKILL(9), SIGSEVG(11), etc.

(2). Origins of signals: Signals to a process originate from 3 sources:
   .Hardware Interrupts: While a process executes, some hardware interrupts are converted to signals delivered to the process. Examples of hardware signals are
      Interrupt key (Control-C), which results in a SIGINT(2) signal.
      Interval timer, which delivers an SIGALRM(14) signal when time expires.
      Other hardware errors, such as bus-error, IO trap, etc.
   .Self-inflicted: When a process in Umode encounters a trap (error), it goes to Kmode to deliver a signal to itself. Examples of familiar trap signals are SIGFPE(8) for floating point exception (divide by 0) and the most common and dreadful SIGSEGV(11) for segmentation fault, etc.
   .From other process by the kill(pid, sig#) syscall, which delivers a sig# signal to a target process identified by pid. The reader may try the following experiment. Under Linux, run the trivial C program
      main(){ while(1); }
   which causes the process to loop forever. From another (X-window) terminal, use ps -u to find the looping process pid. Then enter the sh command
      kill -s 11 pid
   The looping process will die with segmentation fault. The reader may say: that’s incredible! All the process does is executing in a while(1) loop, how can it commit a segmentation fault? The answer is: it does not matter. Whenever a process dies by a signal, its exitValue contains the signal number. The parent sh simply converts the signal number of the dead child process to an error string, whatever that is.

(3). Signals in process PROC: Each PROC has a 32-bit vector, which records signals sent to the process. In the bit vector, each bit (except bit 0) represents a signal number. In addition, it also has a MASK bit-vector for masking out the corresponding signals. A set of syscalls, such as sigmask, sigsetmask, siggetmask, sigblock, etc. can be used to set, clear and examine the MASK bit-vector. A pending signal becomes effective only if it is not masked out. This allows a process to defer processing masked out signals, similar to CPU masking out certain interrupts.

(4). Signal Handlers: Each process PROC has a signal handler array, int sig[32]. Each entry of the sig[32] array specifies how to handle a corresponding signal, where 0 means
(5). Change Signal Handlers: A process may use the syscall
```c
int r = signal(int signal_number, void *handler);
```
to change the handler function of a selected signal number except 9 and 19(SIGSTOP).
The installed handler, if not 0 or 1, must be the entry address of a function in user space
of the form
```c
void catcher(int signal_number){.............}
```

(6). A process may use the syscall
```c
int r = kill(pid, signal_number);
```
to send a signal to another process identified by pid. The sh command
```bash
kill -s signal_number pid
```
uses the kill syscall. In general, only related processes, e.g. those with the same uid, may
send signals to each other. However, a superuser process (uid=0) may send signals to any
process. The kill syscall uses an invalid pid, to mean different ways of delivering the
signal. For example, pid=0 sends the signal to all processes in the same process group,
pid=−1 for all processes with pid>1, etc. The reader may consult Linux man pages on
signal/kill for more details.

(7). A process checks signals and handles outstanding signals when it is in Kmode. If a
signal has a user installed catcher function, the process first clears the signal, fetches the
catcher's address, and resets the installed catcher to DEFAULT. Then it manipulates the
return path in such a way that it returns to execute the catcher function in Umode. When
the catcher function finishes, it returns to the original point of interruption, i.e. from
where it lastly entered Kmode. Thus, the process takes a detour to execute the catcher
function first. Then it resumes normal execution.

(8). Reset user installed signal catchers: User installed catcher functions are intended to
deal with trap errors in user code. Since the catcher function is also executed in Umode, it
may commit the same kind of traps again. If so, the process would end up in an infinite
loop, jumping between Umode and Kmode forever. To prevent this, the Unix kernel
typically resets the handler to DEFAULT before letting the process execute the catcher
function. This implies that a user installed catcher function is good for only one
occurrence of the signal. To catch another occurrence of the same signal, the catcher must
be installed again. For simplicity reasons, we shall adopt this rule in MTX also. However,
the treatment of user installed signal catchers is not uniform as it varies across different
versions of Unix. For instance, in BSD the signal handler is not reset but the same signal
is blocked while executing the signal catcher. Interested readers may consult the man
pages of signal and sigaction of Linux for more details.

(9). Signal and Wakeup: There are two kinds of SLEEP processes in the Unix kernel;
sound sleepers and light sleepers. The former are non-interruptible, but the latter are
interruptible by signals. If a process is in the non-interruptible SLEEP state, arriving
signals (which must originate from hardware interrupts or another process) do not
wakeup the process. If it is in the interruptible SLEEP state, arriving signals will wake it up. For example, when a process waits for terminal inputs, it sleeps with a low priority, which is interruptible, a signal such as SIGINT will wake it up. In MTX, a signal always wakes up a process if it is sleeping or blocked for terminal inputs. It does not wake up those processes if they are waiting for file operations or I/O for block devices.

(10). Proper use of signals: Unix signals are originally designed for these purposes.

. As a unified treatment of process traps: When a process encounters an exception, it traps to kernel mode, converts the trap reason to a signal number and delivers the signal to itself. If the exception occurred in Kmode, the kernel prints a PANIC message and stops. If the exception occurred in Umode, the process typically terminates with a memory dump for debugging.
. To allow processes to handle program errors in Umode by preinstalled signal catchers. This is similar to the ESPIE macro in MVS [IBM MVS].
. Under unusual conditions, it allows a process to kill another process by a signal. Note that kill does not kill a process outright; it is only a “please die” plea to the target process. Why can’t we kill a process outright? The reader is encouraged to think of the reasons. (Hint: the large number of unclaimed bank accounts in Swiss banks).

(11). Misuse of Signals: In many OS books, signals are classified as a mechanism for inter-process communication. The rationale is that a process may send a signal to another process, causing it to execute a preinstalled handler function. The classification is highly debatable, if not inappropriate, for the following reasons.

. The mechanism is unreliable due to possible missing signals. Each signal is represented by a single bit in a bit-vector, which can only record one occurrence of a signal. If a process sends two or more identical signals to another process, they may show up only once in the recipient PROC.
. Race condition: Before processing a signal, a process usually resets the signal handler to DEFault. In order to catch another occurrence of the same signal, the process must reinstall the catcher function BEFORE the next signal arrives. Otherwise, the next signal may cause the process to terminate. Although the race condition could be prevented by blocking the same signal while executing the signal catcher, there is no way to prevent missing signals.
. Most signals have predefined meaning. Indiscriminate use of signals may not achieve communication but confusion. For example, sending a SIGSEGV(11) segmentation fault signal to a looping process is like yelling to a swimmer in the water: “Your pants are on fire!”.

Therefore, trying to use signals for inter-process communication is over stretching the intended purpose of signals. If needed, the message passing mechanism presented in Chapter 6 is a much better way for IPC.
9.4. Implementation of Signals in MTX

9.4.1. Signal Types in MTX

The real-mode MTX uses a 16-bit vector for signals. Therefore, it supports only 15 types of signals, which are the first 15 signals in Unix.

```
#define NSIG 16
#define SIGHUP 1
#define SIGINT 2
#define SIGQUIT 3
#define SIGILL 4
#define SIGTRAP 5
#define SIGABRT 6
#define SIGBUS 7
#define SIGFPE 8
#define SIGKILL 9
#define SIGUSR1 10
#define SIGSEGV 11
#define SIGUSR2 12
#define SIGPIPE 13
#define SIGALRM 14
#define SIGTERM 15
```

In fact, the actual number of signals in the real mode MTX is much less. This is because the Intel x86 CPU in real mode recognizes only a few traps. For instance, it recognizes divide-by-0 and single-step traps by the vector number 0 and 1, respectively, but it does not recognize most other types of traps. For demonstration purpose, other traps are simulated by INT n instructions, where n = 2 to 7. The full set of signals will be used later in protected mode MTX in Chapters 14 and 15.

9.4.2. Signals in PROC Resource

Each MTX PROC has a pointer to a resource structure, which contains the following fields for signals and signal handling.

```
    u16 signal;  // 15 signals; bit 0 is not used.
    int sig[16]; // signal handlers: 0=default,1=ignore, else a catcher in Umode.
```

For the sake of simplicity, MTX dose not support signal masking. If desired, the reader may add signal masking to the MTX kernel.

9.4.3. Signal Origins in MTX

(1). Hardware: MTX supports only the Control-C key from terminal, which is converted to SIGINT(2), and the interval timer of a process, which is converted to INTALRM(14).
(2). Traps: MTX in real mode only supports the divide-by-0 trap. Other traps are simulated by a simu_trap(int n) function, which issues an INT n.
(3). From Other Process: MTX supports the kill(pid, signal) syscall, but it does not enforce permission checking. Therefore, a process can kill any process. If the target process is in the SLEEP/BLOCKED state, kill() normally wakeup/unblock the process.
9.4.4. Deliver Signal to Process

The algorithm of the kill syscall is

```
/************* Algorithm of kill syscall **************/
int kkill(int pid, int sig_number)
{
    (1). validate signal number and pid;
    (2). check permission to kill; // not enforced, may kill any pid
    (3). set proc.signal.[bit_sig_number] to 1;
    (4). if proc is SLEEP, wakeup pid;
    (5). if proc is BLOCKed for terminal inputs, unblock proc;
    (6). return 0 for success;
}
```

9.4.5. Change Signal Handler in Kernel

MTX supports the signal syscall, which changes the handler function of a specified signal. The algorithm of the signal syscall is

```
/*********** Algorithm of signal syscall *************/
int ksignal(u16 sig_number, u16 catcher)
{
    (1). validate sig number, e.g. cannot change signal number 9;
    (2). int oldsig = running->sig[sig_number];
    (3). running->sig[sig_number] = catcher;
    (4). return oldsig;
}
```

9.4.6. Signal Processing in MTX

A CPU usually checks for pending interrupts at the end of executing an instruction. Likewise, it suffices to let a process check for pending signals at the end of Kmode execution, i.e. when it is about to return to Umode. However, if a process enters Kmode via a syscall, it should check and handle signals first. This is because if a process already has a pending signal, which may cause it to die, executing the syscall would be a waste of time. On the other hand, if a process enters Kmode due to an interrupt, it must handle the interrupt first. The algorithm of checking for pending signals is

```
/************ Algorithm of Check Signals ************/
int check_sig()
{
    int i;
    for (i=1; i<NSIG; i++){
        if (running->signal & (1 << i)){
            running->signal &= ~(1 << i);
            return i;
        }
    }
    return 0;
}
```

A process handles outstanding signals by the code segment
if (running->signal)
    psig();

The algorithm of psig() is

/********************** Algorithm psig() **********************/
int psig(int sig)
{
    while(int n=check_sig()){
        // for each pending signal do
        (1). clear running PROC.signal[bit_n]; // clear the signal bit
        (2). if (running->sig[n] == 1)         // IGNORE the signal
            continue;
        (3). if (running->sig[n] == 0)         // DEFAULT : die with sign#
            kexit(n<<8); // high byte of exitStatus=signal number
        (4). // execute signal handler in Umode
            fix up running PROC's "interrupt stack frame" for it to return
            to execute catcher(n) in Umode;
    }
}

9.4.7. Dispatch Signal Catcher for Execution in User Mode

In the algorithm of psig(), only step (4) is interesting and challenging. Therefore, we
shall explain it in more detail. The goal of (4) is to let the process return to Umode to
execute a catcher(int sig) function. When the catcher() function finishes, it should return
to the point where the process lastly entered Kmode. The following diagrams show how
to accomplish these. When a process enters Kmode from Umode, its ustack top contains
an "interrupt stack frame" consisting of the 12 entries, as shown in Figure 9.1.

![Figure 9.1. Process Interrupt Stack Frame](image)

In order for the process to return to catcher() with the signal number as a parameter, we
modify the interrupt stack frame as follows.
(1). replace the uPC (at index 9) with the entry address of catcher();
(2). insert the original return address, uPC, and the signal number after the interrupt
stack frame, as shown in Figure 9.2.

![Figure 9.2. Modified Process Interrupt Stack Frame](image)

This can be done by shifting the interrupt stack frame downward 2 slots, then inserting
the entries (uPC, sig#) immediately after the interrupt stack frame. This changes the
saved usp to usp-4. In a system with stack size limit, the ustack itself may have to be expanded first. Here, we assume that the ustack always has space for the added entries. With the modified ustack, when the process exits kernel, it will return to catcher() as if it had called catcher(sig#) from the place pointed by the original uPC. When the catcher() function finishes, it returns by uPC, causing it return to the original point of interruption. It is observed that in some PC emulators the interrupt stack frame may contain extra entries. For instance, the interrupt stack frame of DOSEMU contains |SP| F800 |uPC|uCS |uflag| rather than |uPC|uCS|uflag| as in a real PC. For such emulators, the interrupt stack frame is meant to contain all the (extra) entries pushed on the stack by the emulator. Otherwise, the above scheme may not work. The virtual machines of QEMU and VMware do not have such problems.

9.5. MTX9.Signal: Demonstration of Signal Processing in MTX

MTX9.signal demonstrates signals and signal processing. In MTX9.signal, when running the sh command, P1 creates a child P2 to run sh and waits for the child to terminate. P2 shows the commands in the /bin directory. The commands include divide, itimer, trap and kill, which demonstrate signals and signal handling. Figure 9.3 shows the screen of running the divide and itimer commands, which are explained below.

![Figure 9.3. Demonstration of Signal Processing in MTX](image)

9.5.1. Divide Trap Catcher
The PC in real mode recognizes divide-by-zero, which traps to the vector number 0. To catch divide-by-zero errors, we can install a trap handler for vector 0 by the INTH macro \_divide INTH kdivide

In the C handler function kdivide(), it sends a SIGFPE(8) signal to the running process. If the process has not installed a catcher, it will die by the signal number 8. Before the divide error, the user may choose to install a catcher for the divide trap. In that case, the process will execute the catcher in Umode. The catcher uses a long jump to bypass the function containing the divide instruction that caused the exception, allowing the process to continue and terminate normally.

9.5.2. Other Traps

The trap.c program issues an INT n (n=2 to 7) to simulate a hardware exception, which sends a signal number n to the process. Similar to the divide trap, the user may install a catcher for the simulated trap n error. In that case, the process will execute the catcher, which uses a long jump to let the process terminate normally.

9.5.3. Interval Timer and Alarm Signal Catcher

In the USER directory, the itimer.c program demonstrates interval timer, alarm signal and signal catcher.

/******************** itimer.c file ********************/
void catcher(int sig)
{
    printf("proc %d in catcher: sig=%d\n", getpid(), sig);
    itimer(1); // set a 1 second itimer again
}
main(int argc, char *argv[]) {
    int t = 1;
    if (argc>1) t = atoi(argv[1]); // timer interval
    printf("install catcher? \[y|n]\n");
    if (getc()=='y')
        signal(14, catcher); // install catcher() for SIGALRM(14)
    itimer(t); // set interval timer in kernel
    printf("proc %d looping until SIGALRM\n", getpid());
    while(1); // looping until killed by a signal
}

In the itimer.c program, the process sets an interval timer of t seconds. Then, it executes a while(1) loop. When the interval timer expires, the timer interrupt handler sends a SIGALRM(14) signal to the process. If the user has not installed a signal catcher, the process will die by the signal. Otherwise, it will execute the catcher once and continues to loop. In the latter case, the process can be killed by other means, such as the Control_C key, which will be shown later when we develop the keyboard driver. In the meantime, we let the catcher() function set another itimer request. When the second itimer expires, the process will die by SIGALRM(14). The reader may modify the catcher() function to install the catcher again. Recompile and run the system to observe the effect.
9.5.4. The Kill Command

The kill command demonstrates the kill system call. To test the kill command, let P1 fork a child (P2) first. Then run the sh process and enter

```
kill pid signal_number
```

which sends a signal to the targeted process, causing it to die. Based on this, the reader may try to develop a program to do inter-process communication using the signal and kill system calls, but watch what would happen if a process repeatedly send the same signal to the target process.

Problems

1. Review Questions:
   (1). Define "interrupts" in your own words.
   (2). What are Hardware interrupts, Process interrupts?
   (3). INT n instructions are usually called software interrupts. What’s the difference between INT n and PROCESS interrupt?
   (4). How does a process get PROCESS interrupts?
   (5). How does a process handle PROCESS interrupts?

2. What is the role of signal 9? Why is it needed?

3. A CPU usually checks for pending interrupts at the end of current instruction. What are the time and places for a process to check for pending signals?

4. Before handling a signal, the kernel usually resets the signal’s handler to DEFault. Why is this necessary?

5. In C programming, a callback function (pointer) is passed as a parameter to a called function, which may execute the callback function. What’s the difference between callback functions and signal catchers?

6. Assume that a process has installed a signal catcher for SIGALRM(14). What if the process exec to a different image? How to deal with this kind of problem?

7. A super user process may kill any other process. An non-super user process may only kill processes of the same user. Implement permission checking in the kill system call.

8. Implement signal masking in MTX.

9. Use SIGUSR1(10) signal to implement Inter-Process Communication (IPC) in MTX. Compare it with message passing by send/recv in Chapter 6.

10. Implement a death-of-child (SIGCHLD) signal. When a process terminates, it sends a SIGCHLD signal to the parent, causing the parent to execute the wait() function.
11. A process gets user inputs to update a counter. Every $t$ seconds it sends the counter value to another process in a message. Design an algorithm for such a process and implement it under MTX. (HINT: use the interval timer and enforce critical regions).

12. Assume that processes communicate by send/recev messages, which are unreliable, e.g. messages may be lost during send/recev. After sending a message, a process expects a reply. If it does not receive a reply within $t$ seconds, it re-sends the same message again. If it receives a reply within $t$ seconds, it must not send the same message again. Design an algorithm for the sending process and implement it in MTX.

References

2. IBM MVS Programming Assembler Services Guide, Oz/OS V1R11.0, IBM
3. Linux: http://www.linux.org
Chapter 9 Figure Captions

Figure 9.1. Process Interrupt Stack Frame
Figure 9.2. Modified Process Interrupt Stack Frame
Figure 9.3. Demonstration of Signal Processing in MTX