Computer Methods and Programs in Biomedicine 26 (1988) 153–158
Elsevier

CPB 00876

Section II. Systems and programs

Multi-tasking control system for real-time processing of biomedical signals

J.A. Westdijk [1], J.A. van Alsté [1] and A.L. Schoute [2]

[1] Biomedical Engineering Division, Department of Electrical Engineering, and [2] Department of Informatics, Twente University of Technology, Enschede, The Netherlands

A general multi-tasking control system has been developed for real-time signal processing. This control system, written in the language PASCAL, enables tasks (expressed as PASCAL procedures) to be performed as separate, concurrent processes, with adjustable priority levels. Modifications of this system such as the addition of new processes and a change of the number of priority levels can be realised easily. The system has been used for the implementation of the real-time algorithms involved in monitoring exercise electrocardiograms. For this application an LSI 11/23 is used with the support of a slave processor for the calculation of inner products. The control system is also suitable for other real-time applications when process requirements are not too heavy.

Multi-tasking control system; Electrocardiograms; Biomedical signal processing; Real-time

1. Introduction

Computer-assisted monitoring of patients based on physiological signals requires real-time data acquisition and processing of the signals involved. Depending upon the complexity and the intensity of the algorithms used, it may become a difficult task to schedule the computer actions that must be performed. Actions may not only be performed in sequences specified in advance but also may be prompted by the actual findings of the partial signal processing. For instance, detection of alarm situations indicates that special activities must be initiated.

For such applications a general multi-tasking control system has been developed. It consists of a kernel of PASCAL routines and enables non-deterministic parallel processing on a single processor system. Real-time activation of processes is one of its features necessary for data acquisition and timing of time-critical actions. The activation or suspension of other processes according to their priority is another one.

The system is applied for the scheduling of processes involved in computer-assisted monitoring of electrocardiograms, obtained during heavy exercise (XECG processing) [1]. The CPU requirements in XECG processing are not too heavy when a separate processor for inner product calculations is available. In this application the control system is suitable for real-time processing at signal sample rates up to 250 Hz.

2. Description of the multi-tasking control system

2.1. General

Real-time XECG processing comprises the handling of various tasks, which differ in complexity, importance and time of activation. So for every task a priority level can be defined, corresponding to its actual urgency. The selection of processes,

Correspondence: J.A. van Alsté, Elektrotechniek, Universiteit Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.

performing the tasks, consequently has to be priority-dependent, in which high-priority processes can interrupt low-priority ones.

According to the activation time a distinction can be made between processes activated on a regular time base (periodical processes) and processes who's activation time depends on the actual signal properties (for example, scheduling of several tasks after each detection of a heart beat) or the activity of another process (for example error messages due to a delay in dispatching a process).

The scheduling and the selection of these types of processes can be realised by a multi tasking control system [2]. Parallel processing on a single processor system can only be realised using a number of queues in which processes are waiting to be served by the CPU or are waiting to be activated under some (time) conditions.

2.2. Queue definition

For the scheduling of real-time processes we installed a 'timer queue', in which processes are sequenced according to their wake-up time. For the execution of processes with different priorities a corresponding number of 'ready queues' is maintained. Processes of which the activation time is dependent upon the actual findings of the partial signal processing can be synchronised by so-called semaphores. They will wait, in case of blocking, in a semaphore queue.

To control the process flow, operations are needed for the insertion of processes in queues and the removal of processes from queues. In Fig. 1 the process flow between the various queues and the corresponding control operations are shown.

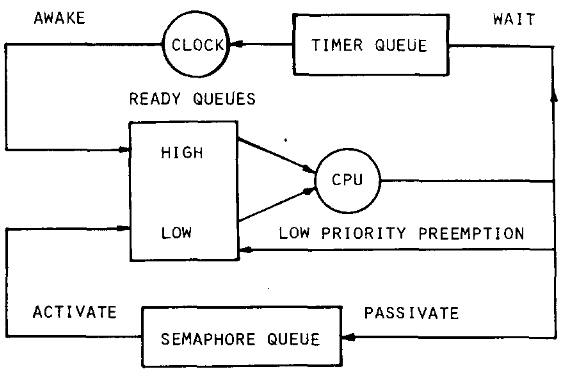

Fig. 1. Process flow.

2.3. Queue operations

In the timer queue, processes are sequenced after being scheduled by a WAIT operation. Real-time scheduling of a process can be relative to a system variable representing the current time or relative to the activation time of another process. Each scheduled process will be activated according to its wake-up time by means of a clock-driven interrupt routine, performing the AWAKE operation. In our application the current time is updated by a clock interrupt every 4 ms. To enable dynamic process control the heading of the WAIT operation has been extended by a priority field. In this way adjusting the priority levels of processes during real-time processing is possible.

Processes can deactivate themselves (from real-time control) by putting themselves in a semaphore queue by means of a PASSIVATE operation. An inactive process blocked by a semaphore can be activated by another process, by means of an ACTIVATE operation. The usage of semaphores for synchronising processes, introduced by Dijkstra [3], has for this application been extended with a time-stamp and a priority field. This enables scheduling of processes relative to an (arbitrary) activation time at adjustable priority levels.

2.4. Activation, preemption and idling

Activation of a process means placing it at the end of the ready queue according to its priority. High-priority processes can preempt low-priority ones served by the CPU, where preempted processes are sequenced at the beginning of the ready queue corresponding to their priority. They are re-activated after the serving of processes with higher priority.

The process with lowest priority is used as an idling loop, which is selected when no other processes are available for execution. Instead of an idling loop some low-priority tasks could be executed. Applied to XECG processing, the lowest priority process is used for keyboard handling.

2.5. Timing and heap / stack pointer managing

To manage processes within queues, processes may be chained by means of a link pointer in their process record. In Appendix 1 the control system declaration part is shown.

The semaphore declaration contains a counter (COUNT) to store the number of activations with respect to that semaphore. Time-stamps corresponding to these activations are stored in a cyclic array (TIME). The size of this array limits the number of pending activations. There are two indices (PROCESSIN, PROCESSOUT), which point to the activation time of the most recent activation and the one which is due to be handled. In Appendix 2 basic queueing operations of processes are shown.

To enable proper switching between 'data environments' of the various processes, each process record contains a stack and heap pointer field. The heap pointer field is also switched in order to maintain the proper heap pointer/stack pointer relationship, which is tested by PASCAL runtime routines. Both pointers get their initial value at the creation of processes.

3. Implementation in OMSI-PASCAL

The multi-tasking procedures are implemented in OMSI-PASCAL [4] on an LSI 11/23 computer. In order to initiate a process a PASCAL procedure which represents the process must have been declared. At the creation of a process a corresponding process record and data stack will be assigned dynamically and the execution of the given procedure will start using these data fields. During execution a process may be blocked, leading to the insertion of the process in one of the process queues and to a switch of process execution. In our application the process bodies contain never-ending loops such that the execution of the tasks will go on indefinitely. The initiation and switching of processes is performed by a number of control system operations (SPAWN, SWITCH, PREEMPT) as shown in Appendix 3.

A creator process (in our case the MAIN program) may start a new process by calling SPAWN. The SPAWN heading contains a procedure parameter (TASK) which specifies the procedure to be performed as parallel process. A noteworthy detail in SPAWN is the copying of the stack-frame corresponding to SPAWN to the stack of the created process.

The selection of the ready processes is handled by another control system operation (PREEMPT, Appendix 3), in which just two levels are used: RUNSTATE, which denotes the priority of the running process, and READYLEVEL, which denotes the highest priority of the ready processes.

By having the READYLEVEL available, the selection of a next process to be served by the CPU is simplified. The actual process switch is performed by the SWITCH operation. During this operation the current heap and stack pointers are stored in the process record of the blocked process and updated according to the selected process.

During control operations no interrupts are allowed. Therefore the body of each control operation is, when necessary, surrounded by interrupt disable/enable statements.

4. Performance

The parallel process kernel for the purpose of XECG processing has been implemented on an LSI 11/23 computer in OMSI-PASCAL, on top of the RT-11 operating system. A memory layout is shown in Fig. 2. The time needed for the control system operations has been measured by displaying the output of a parallel interface (DRV-11) on a logic analyser. The bits of the parallel interface were set or cleared at crucial time points in system routines. In Table 1 the maximum durations measured of the control operations necessary for real-time processing are shown.

In this application eight processes were defined working at four priority levels:
□ two processes with priority level 3 to compute the heart rate and provide the classification of detected beats;
□ one process with priority level 2 to update ECG

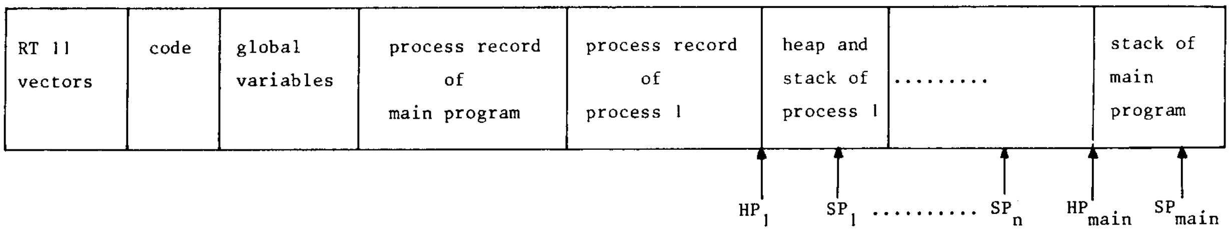
RT 11 vectors
code
global variables
process record of main program
process record of process 1
heap and stack of process 1
....
stack of main program
HP1
SP1
SPn
HPmain
SPmain

Fig. 2. Memory layout.

TABLE 1

Maximum control operation times

Control operation	Maximum duration
Passivate	0.12 ms
Activate	0.26 ms
Awake	0.26 ms
Wait	0.18 ms
Preempt/switch	0.54 ms

beat templates representative for the ECG of the patient being analysed;

□ four processes with priority level 1 for input/output control;

□ one process with priority level 0 for keyboard handling.

In this arrangement real-time processing of an ECG with two lead signals at sampling rate of 250 Hz was possible without problems.

5. Conclusions

Parallel processing on a single-processor system turned out to be successfully applied to real-time processing of electrocardiograms. The advantages of the usage of the described control system are:

(1) Separate description of processes, containing the tasks to be performed. New processes are easily implemented, existing ones can be changed easily.

(2) A simple way of changing the number of priority levels by adjusting the number of ready queues.

(3) The possibility of applying the control system for other real-time processing problems, due to its general purpose design.

The general software approach presented in this paper is an attractive way of real-time process control for cases where CPU requirements are not too heavy, such that the real-time scheduling adequate responses may be obtained.

References

[1] J.A. van Alsté, M.W. la Haye, J. de Vries and H.B.K. Boom, Exercise electrocardiography using rowing ergometry suitable for leg amputees, Int. Rehab. Med., 7 (1985) 1–5.

[2] J.A. van Alsté, A.L. Schoute and H.B.K. Boom, Interactive control of physiological experiments, Comput. Programs Biomed. 18 (1984) 33–40.

[3] E.W. Dijkstra, in: Processing in programming languages, ed. F. Genuys (Academic Press, New York, 1968).

[4] Oregon Software, Manual OMSI PASCAL Version 1.2 for RT-11 (January 1981).

[5] A.L. Schoute and W.A. Vervoort, Realtime concurrent processing with PASCAL, in: Journees d'Electronique, pp. 285–294 (Lausanne Presses Polytechniques Romandes, 1985).

Appendix 1. Control system declaration part

```
CONST maxpri = ... ; (* number of priorities *)
      size   = ... ; (* stacksize *)

TYPE  spawnframe = array[0..6] of integer;

      stack = RECORD
                free : array[0..size] of integer;
                topframe : spawnframe;
              END;

      stackpointer = RECORD
                       CASE integer OF
                        0 : (top : ^spawnframe);
                        1 : (base: ^stack);
                       END;
      process = ^processrecord;

      queue = RECORD
                head,tail : process;
              END;

      semaphore = RECORD
                    count : integer;
                    processin, processout : integer;
                    semchain : queue;
                    time : array[0..9] of integer;
                  END;

      processrecord = RECORD
                        sp,hp : stackpointer;
                        link  : process;
                        timer : integer;
                        prio  : integer;
                      END;

VAR   currprocess, timerqueue : process;
      currtime, difftime : integer;
      readylevel, runstate : integer;
      ready : array[0..maxpri] of queue;
```

Appendix 2. Control operations on queues

```
PROCEDURE activate (VAR s : semaphore);
 BEGIN
  disable;
  WITH s DO
   BEGIN processin := (processin + 1) mod 10;
         time[processin] := currtime;    (* request time
                                            storage *)
         IF semchain.head <> NIL    (* there is a blocked
         THEN                          process *)
          BEGIN
           enter(proc,ready[prio]);
           (* activate process by placing it in the
              ready queue according to its priority *)
           IF readylevel > prio
           THEN (* update level of the
                   highest ready process *)
                readylevel := prio;
           IF prio > runstate
           THEN
            BEGIN
             (* low priority preemption *)
             enterfirst(currprocess,ready[runstate]);
             preempt;
            END;
          END;
         ELSE count := count + 1; (* count number of activa-
                                     tions to be handled *)
   END;
  enable;
 END;
```

```
PROCEDURE passivate (VAR s : semaphore, pri : integer);
 BEGIN
  disable;
   WITH s DO
    BEGIN processout := (processout + 1) mod 10;
          (* index corresponding to the time stamp
             of the activation *)
          currprocess^.prio := pri; (* adjust priority
                                       level *)
          IF count = 0 (* no activations : process blocked *)
          THEN
           BEGIN (* passivate process by placing
                    it in the semaphore queue *)
                 enter(currprocess,semchain);
                 preempt;
           END
          ELSE count := count - 1; (* decrease the number
                                      of activations to be
    END;                              handled *)
  enable;
 END;
```

```
PROCEDURE wait(t, pri : integer);
  BEGIN disable;
        IF t-currtime > 0 (* wake-up time not yet passed*)
        THEN
         BEGIN
           insert(t,currprocess,timerqueue);
           (* insert process in timerqueue
              according to its wake-up time *)
           WITH currprocess DO
            BEGIN timer := t; (* note wake-up time *)
                  difftime := t - currtime
            END;
           IF timerqueue = currprocess (* process is heading
                                          the timerqueue *)
           THEN difftime := t - currtime;
                (* rest-time before next process
                   will be awaken *)
         END;
        enable;
  END;
```

```
PROCEDURE awake;
VAR proc : process;
  BEGIN remove(waitersqueue,proc);   (* remove first
                                        process out of the
                                        waitersqueue *)
        enter(proc,ready[proc.prio]);
        (* insert process in the ready queue, according
           to its priority *)
        WITH proc^ DO
         IF prio > readylevel
         THEN readylevel := prio; (* update level of highest
                                     priority process *)
        IF readylevel > runstate
        THEN BEGIN (* low priority preemption *)
               enterfirst(currprocess,ready[runstate]);
               preempt;
             END;
        IF timerqueue <> NIL (* timerqueue not empty *)
        THEN difftime := timerqueue^.timer - currtime
        ELSE difftime := -1;
  END;
```

```
PROCEDURE clockhandler; (* called by the interrupt
 BEGIN                     routine *)
  currtime := currtime + 1;
  difftime := difftime - 1;
  WHILE difftime = 0 DO awake; (* wake-up process *)
 END;
```

```
PROCEDURE spawn(newprocess : process; procedure task);
VAR currsp, currhp : stackpointer;

BEGIN newprocess := createprocess;
      enter(currprocess,ready[runstate]);
      (* enter creator process in the readyqueue *)

      'store actual value of heap and stack value
       in variables currhp and currsp'

      currprocess^.sp := currsp ; currprocess^.hp := currhp;
      newprocess^.sp.top^ := currsp.top^;
      (* last six words of the main stack are copied to
         the process stack (array TOPFRAME) *)
      currprocess := newprocess;
      currsp := currprocess^.sp;currhp := currprocess^.hp;

      'switch over to new process by copying the values
       of the variables currhp and currsp to heap and
       stackpointer';

       currprocess^.prio := runstate; (* initiate the priority
                                          field *)
       task; (* execution of the task-procedure *)
       preempt; (* termination *)
END;


PROCEDURE preempt;
VAR empty:boolean;
    proc:process;

 PROCEDURE dummy;
  BEGIN
  END;

BEGIN
 runstate := readylevel; (* update runstate according to the
                            process that will be served *)
 remove(ready[readylevel],proc); (* remove ready process to
                                    to be executed *)
 (* update the readylevel *)
 empty := true;
 WHILE readylevel <> -1 DO
  BEGIN empty := ready[readylevel].head = NIL;
        IF empty
        THEN readylevel := readylevel - 1;
  END;
 switch(proc,dummy); (* execute process with priority
                        runstate *)
END;
```

Appendix 3. Example of the application part of the control system

```
PROCEDURE switch (newprocess : process; procedure task);
VAR currsp, currhp : stackpointer;

(* The procedure parameter 'task' has no meaning here. It
   must be added only for the sake of compatibility with
   the procedure SPAWN *)

BEGIN 'store actual value of heap and stackpointer
       in variables currhp and currsp';

      currprocess^.sp := currsp ; currprocess^.hp := currhp;
      currprocess := newprocess;
      currsp := currprocess^.sp ; currhp:=currprocess^.hp;

      'switch over to the new, now current, process
       by copying the values of the variables currhp and
       currsp to heap and stackpointer';
END;


FUNCTION createprocess; (* called by SPAWN *)
TYPE stackptr = RECORD
                CASE integer OF
                 0: (ptr:stackpointer);
                 1: (val:integer);
                END;

VAR newproc : process; newsp : stackptr;

BEGIN new(newproc); (* create new process record *)
      WITH newproc^,newsp DO
       BEGIN new(ptr.base); (* create new stack *)
             hp := ptr;     (* initiate heap pointer at
                               the stack base *)
             val := val + 2*size; (* reset  stack pointer to
                                     to the top frame *)
             sp := ptr; (* initiate stack pointer field *)
       END;
      createprocess := newproc;
END;

(* example application part *)
VAR p1, p2 : process;
    t1, t2, pri1, pri2, delay1 : integer;
    sema1 : semaphore;

PROCEDURE process1;
BEGIN
 REPEAT wait(t1,pri1);
        t1 := t1 + delay1;

        'tasks to perform';

 UNTIL false;
END;

PROCEDURE process2;
BEGIN 'initiate sema1';
 REPEAT passivate(sema1,pri2);
        WITH sema1 DO
         t2 := time[processout]; (* process knows its
                                    activation time *)
        'tasks to perform';

 UNTIL false;
END;

(* MAIN PROGRAM BODY *);

BEGIN new(currprocess); (* create process record
                           for MAIN program *)
      currtime := 1; difftime := -1;
      runstate := 0; readylevel := 0;
      initiate(clock);
      spawn(p1,process1);
      spawn(p2,process2);

      REPEAT UNTIL false; (* idle loop with lowest priority *)

END.
```