Software—Concepts and Tools (1996) 17: 6–12

Software—Concepts and Tools

Tasks versus Threads: An Alternative Multiprocessing Paradigm

Niklaus Wirth

ETH Zürich, Institut für Computersysteme, CH-8092 Zürich, Switzerland
e-mail: wirth@inf.ethz.ch

Abstract. An alternative to threads is presented as a paradigm for single-processor multi-tasking systems. It avoids complex and hidden mechanisms for process scheduling, and is therefore particularly suitable for real-time systems requiring fast response times, and for small systems in general. The essence of the alternative is to base processes (called *tasks*) on subroutines instead of coroutines. Tasks are classified according to their priority. A task may preempt, i.e., temporarily suspend, any task of lower priority. However, apart from temporary suspension, tasks run to completion. As a consequence, such a system operates with a single workspace (stack), on which workspaces of interrupted tasks are stacked. This makes the system's use of storage highly economical and efficient. We present an implementation of this concept requiring a few changes only to the basic Oberon System.

Keywords: multiprocessing discipline, real-time, thread, Oberon, task priority

Background

Most modern computing systems include some support for multi-tasking. Programming languages feature constructs to express concurrency, and operating systems include routines to allocate resources to parallel processes. Since parallelism introduces a considerable amount of complications and pitfalls unknown in conventional programming, it is appropriate to ask for reasons for and justifications of parallelism.

Parallelism was first introduced into programming languages to model and simulate systems characterized by multiple actors whose behaviour could be described as sequences of discrete events. These patterns of behaviour can be mapped directly into programs called parallel processes. Any problems arising from the fact that they are "parallel" are not associated with programming, but rather with the modelled system. The task of the underlying operating system is merely to allocate resources (memory, processors) to the processes. If the number of processors—usually 1—is smaller than that of processes—typically many—the former are time-shared among the latter. Real time and system time are distinct entities.

The second reason for introducing parallelism was the emergence of systems with multiple processors in large computing systems to be utilized by multiple users in the most effective way. Also, input/output devices became regarded as concurrently active processors. Optimal scheduling of processes and optimal allocation of processors became an important and difficult problem in operating system design. Many of the concepts of simulation systems could be adopted.

The third reason for parallelism became relevant only recently. It is simply the demand for extreme speed of computation made possible by systems with a large number of identical processing elements for performing a single, common task. Here, genuinely concurrent processing is present, whereas in the former cases concurrency is mostly simulated with the intent that the programmer may view the processes as being executed simultaneously, as if the computer was equipped with as many processors as there are processes.

In this paper, we will focus on systems belonging to the second category, where the number of processors is smaller than that of processes. In fact, we will consider single-processor systems only. Is a further consideration of such systems at all justified in view of the ready availability of cheap processors? The answer is clearly affirmative in view of the flexibility of software compared to that of dedicated special hardware. And after all, in most cases the power of a single processor is (more than) adequate.

Thus, systems to be considered here are characterized by the switching of a processor among processes. Since switching typically occurs quite frequently, it must be fast. Early multi-tasking systems were designed to run processes (jobs) submitted by different

users. The system must guarantee the absence of interference of processes, even under the presence of programming errors in the user programs. The necessary safety measures turn out to be a serious impediment to fast switching. Hardware provides address mapping, letting every programmer view the program to be executed in a memory disjoint from those of all other tasks. Therefore, task switching must include the reloading of registers used for address mapping.

As a result, the switching overhead is intolerably large for all purposes except for switching among disjoint jobs in multi-user systems, which appear as somewhat old-fashioned in the era of personal computers. Switching among processes within the same address space, however, can be implemented much more efficiently. Such processes are known as light-weight processes or *threads* [2]. Like a (heavy-weight) process, a thread is a single, sequential flow of control, and it is implemented as a coroutine. The thread represents a much more convenient abstraction than the coroutine, because in the latter the destination of a process switch must be explicitly specified as a coroutine jump. Coroutines display a character like jumps. Processes abstract from coroutines in the same way as for example while statements abstract from jumps.

With threads, the destination of a switch is implied in synchronization operations. A possible set of such operations are the *signal* and *wait* primitives operating on conditions [4], or the P and V primitives operating on semaphores [3]. These primitives imply some overhead to determine the destination of the underlying coroutine jump, i.e., for the implicit processor scheduling.

Most modern systems feature threads as a programming facility, and their indispensability has been widely accepted [1]. However, threads have also given rise to some curious phenomena. For example, the inquisitive user of a workstation may find that in the course of a session dozens if not hundreds of threads had been generated which continue to coexist without apparent reason. Even if they remain idle, they burden a system considerably, because every thread requires its own workspace, a stack of procedure activation records. If stack size is (arbitrarily) fixed, this represents a waste of memory space, and if stack size is determined dynamically on demand, the overhead in time is quite heavy. The thread facility seduces programmers to its undeliberate use by abdicating responsibility to hidden scheduling and allocation mechanisms. We suspect the thread to be a contributor to fat software [6].

So, we have been looking for an alternative to threads, in spite of their popularity. The alternative must be more transparent, free of implicit run-time support of unknown complexity. It must allow extremely fast process switching with an overhead comparable to that of a procedure call. As a result, the alternative should be attractive in particular to real-time applications, where switching time must be at most a few microseconds. We propose to use the old concept of the interrupt handler in place of the thread. Let us investigate the consequences.

Subroutines in Place of Coroutines

Let the following be the boundary conditions for the alternative paradigm to be investigated:

- The system has a single processor.
- The system is primarily used by a single operator pursuing various tasks.

Under these conditions, the following observation holds:

- In the absence of real-time constraints, there is no reason to consider parallelism.

Indeed, the various tasks pursued by the operator may just as well be tackled sequentially, one after the other. No speed is to be gained by introducing pseudo-parallelism. On the contrary, frequent task switching is counterproductive. This observation has been the basis of the tasking concept of the Oberon system [5, 8]. Its core contains an infinite loop in which input devices are polled. If an input is sensed, a corresponding task handler (a procedure called a *command*) is called, and it runs to *completion*. Hence, task switching occurs implicitly after task completion and return to the central loop. A single workspace is sufficient.

However, postulating the absence of real-time requirements is problematic as soon as a human user is involved. The operator may well associate different degrees of urgency to the tasks at hand. A stroke of the keyboard should cause the character to be displayed immediately; a query to a data base may take some time. We conclude:

> Introducing concurrency is justified only if tasks have distinct priorities.

Priorities have discrete values, and there is a fixed (small) number of priority levels. We may thus characterize the alternative paradigm as follows:

> Tasks are executed on each priority level in sequential order, one after the other. The occurrence of a condition, an event, introducing a task of higher priority, temporarily suspends execution of the current task, which is resumed after the higher-

priority task has been completed. We say that tasks may be *preempted.*

What is the right number of priority levels? In our case study, which is a version of the Oberon system, we have opted for three. First, because all concepts and problems occuring with more levels are present with 3 levels, and second, because the number 3 can easily be justified: On the first level, we find tasks of zero urgency, and we therefore call them *background* tasks. On the highest level reside tasks associated with input and output devices requiring immediate response. We call them *real-time* tasks. The middle level contains the tasks requiring medium speed response, such as the handling of input from the human user working with keyboard and mouse.

0. Background tasks, no priority.
1. Interactive tasks, preemptible only by real-time tasks.
2. Real-time tasks, not preemptible.

Tasks versus Threads

The advantage of tasks over threads is the gained simplicity in implementation and transparency in use, as there is no hidden scheduling algorithm involved that might produce unexpected behaviour or unpredictable performance. Naturally, there is also a price to be paid. Indeed, the coin might be flipped, and the fact that threads remove processor scheduling from the programmer's considerations may be perceived as an advantage. Threads in fact constitute a convenient abstraction on a higher level. They let repetitive processes be expressed as coherent programs interacting with other processes through interchange of signals and messages. In this way, processes are abstracted from concrete processors, and their mapping in time onto available processors is left to the implementation at hand.

The fundamental difference between tasks and threads is that the latter retain a state when suspended, whereas task do not. This makes it necessary to assign a private workspace to every thread. The task, on the other hand, runs to completion and has no state when terminated. If required, a thread can be represented by a set of tasks, each task standing for that part of the thread which lies between a pair of consecutive breakpoints. The thread's state is then to be held by global variables. This should be the exception, however, and if it occurs frequently in a design, then threads are probably the more appropriate abstraction to express the case. The paradigm of tasks is preferable in relatively simple systems and has the advantage of being fast. Hence, a likely case for its beneficial application are real-time control systems, data acquisition systems, embedded systems, smaller systems in general, where economy of storage and time are of paramount importance.

Task Creation and Activation

Every real-time task is associated with a device. It is created—the technical term being *installed*—by calling *Install(P, devno)*, where *P* is the (parameterless) task procedure and *devno* is a number associated with the device. P is activated by an *interrupt signal* issued by the device. This signal interrupts the task currently being served and switches the processor to the real-time task.

The set of interactive tasks is represented by a list of task descriptors, each specifying a task handling procedure. Tasks are created by inserting a descriptor T into this list by calling *Install(T)*, and they can be removed from the list by calling *Remove(T)*. The tasks in this list are executed sequentially and periodically in a fixed time interval. Every task handling procedure has the form

```
IF condition THEN action END
```

The task action is executed not in each time interval, but only when the program-specified activation condition is met. A typical time interval is 20—40ms, implying reactivation 25—50 times per second.

Background tasks are also represented by a list of descriptors. A task is submitted for execution by inserting a descriptor, and it is automatically removed from the list upon completion. Background tasks are inserted by the operator activating (mouse clicking) the command

```
Submit P x0 x1 ...
```

written in some window (viewer), where P is a procedure (command) name and x0, x1, ... are P's parameters. In contrast to interactive tasks, background tasks are executed only once, and not repeated until removed. The background facility may be regarded like a batch processing operation where jobs are submitted and executed in the sequence of their submission.

Task Interaction

So far, we have ignored potential problems arising from task interaction. Every facility for synchronization and communication between tasks incurs a certain overhead. Worse, however, is the fact that tasks may be blocked. Blocking in a single-processor system requires task switching, an action that we have explicitly excluded by postulating that tasks run to completion, and as a consequence of

replacing coroutines (with suspension and resumption points) by subroutines (procedures). At this point, our attempt to find an alternative paradigm to threads seems to be doomed to failure. Let us therefore find the origin of the presumed difficulties.

A widely accepted paradigm for synchronization and communication in multi-tasking systems is message passing. Here, data exchange implicitly also serves for synchronization. The alternative is the use of global, shared variables, which is the obvious solution in the case of shared memory systems, and in particular in the case of single-processor systems. Here, we differentiate between variables shared for synchronization (such as general semaphores) and those for protecting against unwanted interference when accessing shared variables (such as binary semaphores). Let us consider the problem of mutual exclusion first.

A facility for mutual exclusion satisfying the requirements of structured design is the *monitor* [4]. It may be considered as a module containing a number of procedures operating on a shared resource. What makes the module a monitor is that every procedure is enclosed in a pair of lock and unlock statements operating on a hidden binary semaphore associated with the monitor, thus guaranteeing that at most one process may operate on any of the procedures in the monitor, thereby providing interference-free access to the monitor's variables. Here we recall that we focus on single-processor systems. An attempt of the processor to reenter a monitor can only arise if a process executing a monitor procedure is interrupted, and if the interrupting process calls a procedure of the same monitor. This case can easily be prevented by replacing the semaphore operations by operators disabling and enabling interrupts, fast operations available in every computer's instruction set. The interrupt enable bit (typically contained in a status register) may be regarded as a global lock semaphore, and its use may be judged as a drastic and rather indiscriminate solution. However, it becomes acceptable considering that monitor operations are very short and fast. An improved solution consists of disabling only those interrupts that may cause reentering of the particular monitor. It requires an appropriate hardware support in the form of a *priority interrupt system* as present in many process control computers of older provenance. Such systems do not shut out all interrupts when one is accepted, but only those of lower priority.

Although our solution to the mutual exclusion problem appears as straight forward, caution is recommended. Operating systems are inherently managers of shared resources to which access is requested very frequently. If every access to the storage manager and every access to a file—files are global when registered in a directory—causes interrupts to be shut out, the functioning of the overall system is impaired, and in particular real-time response limits can no longer be guaranteed.

If neither protection against mutual interference nor efficiency and fast response are to be sacrificed, the only option remaining is to postulate a number of restrictive rules about accessing resources, that is, some sacrifice of generality. We propose the following set of rules as a basis, and have found that except for pathological constructions they do not represent any serious hinderance to programming and system design.

1. Real-time task (level 2) refrain from using
   - dynamic storage management (procedure NEW)
   - file and file directory access
   - window management
2. Background (level 0) tasks do not make use of display and window management. Instead, the system provides a specific log text displayed in a window exclusively reserved for background tasks.
3. Files cannot be changed (written) once they have been registered in the directory.

Rule 1 bars real-time tasks from using global resources except global variables explicitly declared for the task's communication with other tasks. The rule's benefit is that when lower priority tasks access global facilities, only the interrupts activating interactive tasks (level 1) must be disabled, thus allowing real-time interrupts to be still serviced. Evidently, we postulate a two-level priority interrupt scheme.

Rule 2 has the effect that the display areas used by interactive tasks are disjoint from that used by background tasks. As they are statically disjoint, no provision is needed at run-time against interference . The frequent display operations need not involve any locks.

Rule 3 prevents writable files to appear under different variable names. If a file is accessible under different names, the danger is that it becomes inconsistent. For example, one task associating a file with variable F, while another task associates the same file with variable G, might lead to data inconsistency unless writing of F and G were prohibited.

```
F := Files.Old("Agenda.Text"); ...
G := Files.Old("Agenda.Text"); ...
```

The benefit of Rule 3 is that no special precautions are required to assure file consistency. In particular, write operations need not involve locks.

As a consequence of the imposed restrictions, it suffices to disable interrupts activating interactive tasks in the following cases:

- access to storage management (NEW, GC)
- access to file directory (Search, Insert, Delete)

It is quite obvious that the postulated rules, although no serious impediment to system design, drastically reduce the machinery necessary to exclude unwanted interference among processes.

Task Synchronization

Synchronization and communication between tasks is achieved in our model through the use of global, shared variables. Synchronization can be achieved through general semaphores represented by the INC and DEC operators acting on integers, assuming that these two operations are atomic, i.e., cannot be interrupted. The following example stands for the frequent and typical case of sequential data buffering with a finite, cyclic buffer B. n is the number of elements in B, and it is the only critical variable. No mutual exclusion guard is required, since n is changed by the atomic INC and DEC operators only. Variables in and B[in .. out-1] "belong" to the producer (Handler), out and B[out .. in-1] to the consumer (Read), indices taken modulo N.

```
MODULE Input;
  IMPORT SYSTEM, Kernel;
  CONST N = 64;  (*buffer capacity*)
  VAR n*: INTEGER;  (*0 <= n <= N*)
    in, out: INTEGER;  (*buffer indices*)
    B: ARRAY N OF CHAR;  (*buffer*)
    overrun*: BOOLEAN;

PROCEDURE+ Handler;
BEGIN (*real-time task*)
  IF n < N THEN
    SYSTEM.GET(devAdr, B[in]);
    in := (in+1) MOD N; INC(n)
  ELSE overrun := TRUE
  END
END Handler;

PROCEDURE Read*(VAR ch: CHAR);
BEGIN (*no mutex required*)
  REPEAT (*idle*) UNTIL n > 0;
  ch := B[out]; out := (out+1) MOD N; DEC(n)
END Read;

BEGIN (*init*)
  n := 0; in := 0; out := 0; overrun := FALSE;
  Kernel.Install(Handler, chanNo)
END Input.
```

When this module is loaded, a real-time task called *Handler* is installed. It is reactivated whenever the associated device can deliver a next character. Procedure *Read* is called from lower-priority tasks to fetch the characters from the buffer.

A similar but presumably more sophisticated example is a network interface, where the level-2 task (interrupt handler) accepts arriving packets under stringent real-time constraints and deposits them in a buffer. The consumer is typically another task of level-1 priority, i.e., a recurring task, whose activation condition is "buffer nonempty". The sender of packets must lock out all interrupts in order to comply with timing specifications for synchronous data transmission. Locking and unlocking is specified by calling a new predeclared procedure LOCK(TRUE) and LOCK(FALSE).

Our third example shows how "busy waiting" for responses from peripheral devices is avoided. Here, a document file is to be printed. We propose the following scheme: Let printing jobs be represented by descriptors in a queue, each denoting a file. Let print-task A be classified as a level-1 task with activation condition "print queue nonempty". It picks the first descriptor from the print queue and opens the denoted file. It then replaces itself with print task B in the list of level-1 tasks. Task B's activation condition is "printer ready". It reads data from the file and generates the bitmap for the image of the first page and terminates by triggering the printing engine. Now the processor is free for other tasks. It returns to the printing activity through a timer interrupt, and when scanning level-1 tasks finds the printer to be idle. Then the next page is processed, and this proceeds until the last page has been handled, whereupon task B reinstalls task A, which picks the next print job from the queue, if there is one. Note that no background tasks are involved in this example either.

An Implementation

The described model has been implemented on the same computer (Ceres-2 with NS32000 processor) which had originally served as platform for the Oberon system. The adaptation required surprisingly few changes.

Tasks of level 2 are implemented as parameterless procedures using an unpublished compiler option. This option (indicated by a + sign following the symbol PROCEDURE), causes (1) the saving of some registers upon entry and their restoring upon exit, and (2) to end the code with a *return interrupt* instead of *return subroutine* instruction. Procedure *Kernel.Install(P, n)* assigns P's address to the computer's interrupt vector with index *n*.

Tasks at level 1 are essentially keyboard and mouse handlers, apart from additional ones to be provided by the system's users. The set of tasks is triggered by a timer interrupt (*tick*) once every 40ms. This scheme has the advantage that—in the absence of a priority interrupt system—all interactive tasks are interrupt protected by a single instruction stopping the timer. The timer interrupt handler is shown below:

```
PROCEDURE Tick;
  VAR x, y: INTEGER; keys: SET; ch: CHAR;
    M: InputMsg; V: Viewers.Viewer; T: Task;
BEGIN
  IF Input.Available() > 0 THEN (*keyboard task*)
    Input.Read(ch);
    M.id := consume;
    M.ch := ch; FocusViewer.handle(FocusViewer, M)
  END ;
  Input.Mouse(keys, x, y);
  IF (keys # {}) OR MouseMoved THEN (*mouse task*)
    V := Viwers.This(x, y); M.id := track;
    M.x := x; M.y := y; V.handle(V, M)
  END ;
  T := NextTask;
  WHILE T # NIL DO (*other level-1 tasks*)
    IF T.time < Time() THEN T.handle END ;
    T := T.next
  END
END Tick
```

FocusViewer is a global variable designating the viewer (window) holding the keyboard focus visualized by the caret. *NextTask* is a global variable representing the head of the list of interactive tasks. Each task is represented by a descriptor:

```
Task =      POINTER TO TaskDesc;
TaskDesc =  RECORD
              handler: PROCEDURE;
              time: LONGINT;
              next: Task
            END
```

A task is activated only if the value of its field *time* is less than the current time provided by the timer. Thereby the programmer may determine the frequency by which a task is reactivated. In the Ceres system, the maximum frequency is set to 25 Hz, and the timer value increments every 3.3ms. Hence, it is possible to implement interactive tasks requiring frequent attention such as keyboard input, mouse handling and cursor update, as well as tasks to be reactivated rarely, such as a clock update every minute.

Interactive tasks are installed and deleted by the following system procedures:

```
PROCEDURE Install* (T: Task);
  VAR t: Task;
BEGIN t := NextTask;
  IF t # NIL THEN
    WHILE (t.next # NIL) & (t # T) DO
      t := t.next END;
    IF t # T THEN t.next := T END
  ELSE NextTask := T
  END
END Install;
PROCEDURE Remove* (T: Task);
  VAR t: Task;
BEGIN t := NextTask;
  IF t # NIL THEN
  IF t = T THEN NextTask := t.next
  ELSE
    WHILE (t.next # NIL) & (t.next # T) DO
      t := t.next END;
    IF t.next = T THEN t.next := T.next END
  END
  END
END Remove;
```

Tasks at level 0 (background) are activated by the Oberon system's central loop, which receives control when all interrupt-triggered task are completed. Tasks are picked from the fifo queue whose root is *NextBTask*:

```
LOOP T := NextBTask;
  IF T # NIL THEN NextBTask := T.next; T.handle END
END
```

Our implementation differs slightly from this obvious and expected solution. It takes into account that background tasks are typically submitted by the workstation's operator invoking (clicking) a submit command of the form

```
System.Submit  M.P  x0 x1 ... xn
```

Hence, the background task descriptor does not specify a procedure (handle), but rather the text following "System.Submit", from which not only the task's body (M.P) but also its parameters can be derived.

```
BTask =      POINTER TO BTaskDesc;
BTaskDesc =  RECORD next: BTask;
               text: Texts.Text;
               pos: LONGINT
             END

PROCEDURE Loop*;
  VAR this: BTask; S: Texts.Scanner;
BEGIN (*background process, central loop*)
  LOOP
    IF ActCnt <= 0 THEN
      Kernel.GC; ActCnt := BasicCycle END ;
    this := NextBTask;
    IF this # NIL THEN
      NextBTask := this.next; Call(...) END
  END
END Loop;

PROCEDURE Submit* (T: Texts.Text; pos: LONGINT);
  VAR Q, q: BTask;
BEGIN
  NEW(Q); Q.next := NIL; Q.text := T;
    Q.pos := pos; q := NextBTask;
  IF q = NIL THEN NextBTask := Q
  ELSE
    WHILE q.next # NIL DO q := q.next END ;
    q.next := Q
  END
END Submit;
```

Summarizing, the adaptation of the Oberon System to the described multitasking paradigm consists of the following steps:

0. Global variables are localized, except when representing genuine global resources (such as the root of the viewer structure, the root of the file directory, etc.) or when representing global status (such as *CurFont*, *CurColor* etc.).
1. A timer interrupt is introduced to trigger interactive tasks periodically.
2. The body of the central loop is moved to the timer interrupt handler.
3. The central loop is changed to handle background tasks.
4. Mutual exclusion guards are inserted in certain global procedures of storage, viewer and file directory management.

Conclusions

Tasks can be used as alternative to threads in the formulation of concurrency. They are procedures that run to completion except when interrupted (temporarily suspended) by a task of higher priority. In contrast to threads, which are based on the coroutine concept and require private workspaces, tasks share a single stack as their workspace. Since task switching is triggered by interrupt signals, no complex, hidden scheduling mechanism is required, resulting in quick response and low overhead. The task concept therefore appears as particularly attractive for real-time systems with stringent timing requirements, and for small systems in general, such as embedded applications, where economy of storage and time are important. An adaptation of the Oberon operating environment has demonstrated the viability of the concept. Tasks, called commands, are a conceptual pillar of the Oberon operating environment. Making them interruptible adds significant power to the concept and is shown to require surprisingly few changes and additions to the existing system, hence retaining its basic compactness and efficiency.

References

1. Andrews GR, Schneider FB (March 1983) Concepts and Notations for Concurrent Programming. ACM Comp. Surv., 15, 1, 3–43
2. Birrell AD (1989) An Introduction to Programming with Treads. Digital, Systems Research Center, Palo Alto, Report No. 35
3. Dijkstra EW (September 1965) Cooperating Sequential Processes. TU Eindhoven
4. Hoare CAR (October 1974) Monitors: An Operating Systems Structuring Concept. Comm. ACM 17, 10, 549–557
5. Reiser M (1991) The Oberon System. Addison-Wesley
6. Wirth N (February 1995) A Plea for Lean Software. IEEE Computer
7. Wirth N, Gutknecht J (September 1989) The Oberon System. Software—Practice and Experience, 19, 9, 857–893
8. Wirth N, Gutknecht J (1992) Project Oberon. Addison-Wesley