SHORT-TERM SCHEDULING IN MULTIPROGRAMMING SYSTEMS

Per Brinch Hansen
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Abstract

This paper defines a set of scheduling primitives which have evolved from multiprogramming systems described by Dijkstra, Lampson, Saltzer, and the present author. Compared to earlier papers on the same subject, the present one illustrates a more concise description of operating system principles by means of algorithms. This is achieved by (1) describing the primitives as the instructions of an abstract machine which in turn is defined by its instruction execution algorithm; (2) introducing a notation which distinguishes between the use of synchronizing variables (semaphores) to achieve mutual exclusion of critical sections, and to exchange signals between processes which have explicit input/output relationships; (3) considering the influence of critical sections on preemption and resumption; and (4) using a programming language, Pascal, which includes natural data types (records, classes, and pointers) for the representation of process descriptions and scheduling queues. The algorithms are written at a level of detail which clarifies the fundamental problems of process scheduling and suggests efficient methods of implementation.

KEYWORDS AND PHRASES: multiprogramming systems, operating systems, concurrent processes, process synchronization, semaphores, critical sections, priority scheduling, preemption and resumption, Pascal.

CR CATEGORIES: 1.52, 4.2, 4.3

1. Background

At an early stage in the design of the multiprogramming system for the RC 4000 computer[1] a distinction was made between those sequential processes which control input/output and those which perform computations. This distinction, which was based on differences in process scheduling and storage addressing, had a drastic influence on the real-time characteristics of the system. On the one hand, input/output processes could be initiated immediately by interrupts and run without preemption for several milliseconds. On the other hand, due to the use of fixed round-robin scheduling, computational processes could only respond to urgent, external events in 10-100 milliseconds. The extendability of the system was also strongly affected by this decision. The input/output processes enjoyed special privileges of addressing which enabled them to enter a global critical section and execute shared procedures. However, the smallest modification of any of them required reassembly and testing of the entire system nucleus. In contrast, computational processes were unable to share procedures but were easy to implement and test separately. The system nucleus was indeed built to create, execute, and terminate computational processes dynamically. This problem was caused partly by the addressing characteristics of the computer and partly by inadequate understanding of the issues involved.

In the following, I describe a model of a multiprogramming system in which this mistake is avoided by treating all processes in a uniform manner at the most elementary level of scheduling. The paper serves two purposes: it illustrates a concise description of operating system principles by means of algorithms, and it defines a set of scheduling primitives which have evolved from recent multiprogramming systems described by E. W. Dijkstra, B. W. Lampson, J. H. Saltzer, and myself[1-4]. The primitives are regarded as the instructions of an abstract processor which is defined by its instruction execution algorithm. The algorithms are written at a level of detail which clarifies the fundamental problems of process scheduling and suggests efficient methods of implementation.

The description language used is Pascal created by N. Wirth[5]. In the words of C. A. R. Hoare: "This is a language which gives equal attention to the methods of structuring data and of structuring program. It is a very simple language, requiring a compiler as fast and as small as a good Algol 60 implementation, and producing machine code of an efficiency and compactness typical of a good Fortran compiler. Treatment of programming errors is excellent, fully adequate for its use as a teaching tool." Pascal is easily understood by programmers familiar with Algol 60. Pascal, however, is a far more natural tool for the description of operating systems than Algol 60 due to the inclusion of data structures of type record, class, and pointer.

2. Short-Term and Medium-Term Scheduling

Our intellectual inability to analyze all aspects of a complex problem in one step forces us to divide the scheduling problem into a number of decisions which are made at different levels of programming. The view of scheduling presented here recognizes two main levels. At the lower level, which may be called hardware management or short-term scheduling, the objective is to allocate physical resources to processes, as soon as they become available, to maintain good utilization of the equipment. This level of programming simulates a virtual machine for each sequential process and a set of primitives which enable cooperating processes to achieve mutual exclusion of critical sections and communicate with each other.

At the higher level of scheduling, which may be called user management or medium-term scheduling, the aim is to allocate virtual machines to users according to the rules laid down by system management. Typical tasks at this level are to establish the identity and authority of users; to input and analyze their requests; to initiate and control computations; to perform accounting of resource usage; and to maintain system integrity in spite of occasional malfunction of the hardware;

I owe the terms short-term and medium-term scheduling to C. A. R. Hoare. The distinction between these levels was clearly made in the thesis by J. H. Saltzer.[4]

The ability to experiment with scheduling policies at the user level is a vital requirement of extensible operating systems. It will be shown that a suitable choice of data structures and primitives at the short-term level enables sequential processes to schedule other processes according to any strategy desired at the medium-term level (within the given technological limits). From the RC 4000 system, we learned that the decision about whether a certain process has the right to schedule another process should be separated from the decision about how the scheduling is done. The rules of protection among processes are subject to experimenting just like the rules of scheduling. Processes should not be born with certain rights of access to shared objects, but should earn them by demonstrating their own usefulness and correctness. Consequently, I make no distinction in the following between

processes which are part of operating systems and processes which are part of user computations.

3. Process States and Scheduling Primitives

I consider a computer system in which a limited number of identical processors are multiplexed among a (possibly larger) number of sequential processes. The processors are connected to a single, internal store which is assumed to be large enough to satisfy all processes at any time. The short-term problem of scheduling a limited internal store using a larger backing store is not discussed here.

At the short-term level of scheduling, a process is either running on a processor or waiting in a queue. A process can wait for an idle processor or for a timing signal from another process. In the former case, the process is said to be ready, in the latter, it is blocked. The synchronizing primitives for the latter case are the by now familiar P and V operations on semaphores introduced by E. W. Dijkstra[2]. I use the following notation to distinguish between the use of semaphores to (1) achieve mutual exclusion of critical sections, and (2) exchange signals between processes which have explicit input/output relationships:

```
(1)  critical v do S
(2)  wait(v) ... signal(v)
```

where v is a variable of type semaphore and S is a statement. The structured statement (1) clearly shows that the semaphore is used to label S as a member of the set v of statements which must exclude each other in time. It forces the programmer to use P and V operations strictly in nested form just as the begin and end of compound statements. It prevents the undisciplined use of P and V operations frequently seen, e.g.,

```
P(v);
.....
if B then begin V(v); P(w); end
     else V(v);
```

which to my taste is like terminating a compound statement by a conditional end deliminter! One is tempted to use this style of programming because the well-structured alternative requires the use of a local boolean b to transmit the value of expression B (involving shared variables) outside the critical section, e.g.,

```
critical v do
begin ..... b:= B; end
if b then wait(w);
```

This minor point of efficiency, however, can be solved by the disciplined use of jumps as we shall see later.

The scheduling of processes in a given queue should reflect the policy of system management towards the type of work performed by the processes or the group of users responsible for them rather than their random order of arrival in the queue. Hence, we assume that a priority number is assigned to each process and that the queues are ordered accordingly. However, a compromise must be made to avoid spending excessive amounts of processor time inspecting and rearranging queues. The present model assumes that priorities remain fixed over the intervals of time considered at the short-term level of scheduling.

Dynamic priorities are obtained by stopping the execution of processes, assigning updated values to their priority variables, and starting them again. Thus, we are led to the following scheduling primitives:

```
stop(p)
start(p, n)
```

where p and n are variables of type process description and priority number, respectively. Two additional primitives, which create and terminate a process in the stopped state, are not discussed here.

A graph of the process states and the possible transitions between them is shown in figure 1.

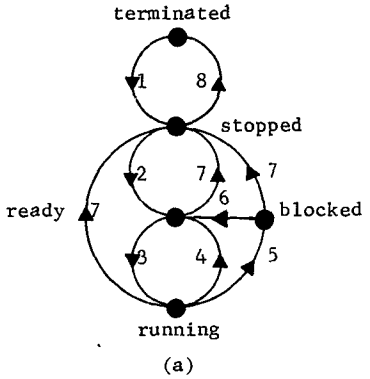

Fig. 1. A graph representation of the process states (a), and the primitives that cause the transitions between them (b).

The primitives initiate and examine are only executed as part of the instruction execution algorithm of a processor. The primitives create, start, wait, signal, stop, and terminate, however, can be used by processes to schedule other processes at the medium-term level.

4. Process Descriptions and Queues

The set of sequential processes will be represented in Pascal by the following data structure:

```
var process set: class maxnum of
                 record pred, succ: ↑process set;
                        state: statename;
                        priority: integer;
                        program pointer: address;
                 end
```

This declaration introduces a class of records. Each record describes a process by its state, priority, and program pointer. When the process is waiting in a queue, its record is linked to the records of the predecessor and successor processes in the same queue.

Note, that a declaration in Pascal consists of an identifier of a variable followed by its type, e.g.,

```
priority:  integer;
```

The type integer is predefined. Other scalar types can be defined by enumerating a set of values (or, more precisely, a set of identifiers denoting constants), e.g.,

```
type statename = (stopped, waiting, running)
```

or by defining a range of values, e.g.,

```
type address = 0..65535
```

The components of a class are referenced through variables of type pointer. Each pointer variable is by its declaration bound to the components of a particular class, e.g.,

```
var process:  ↑process set
```

Initially, a class is empty and its associated pointer variables all have the value nil.

The standard procedure

```
alloc(process)
```

creates a new record in the process set and assigns its pointer to the variable process. The record fields can now be referenced through the pointer variable using the notation

process↑.pred ... process↑.state ...

The record referenced by the pointer variable process can be removed by means of the standard procedure

reset(process)

which also assigns the value nil to the pointer variable.

The class concept closely mirrors the dynamic creation and termination of process descriptions. The value of a pointer variable can be assigned to other pointer variables bound to the same class. This is used here to implement a scheduling queue as a doubly-linked list of process records as shown in figure 2.

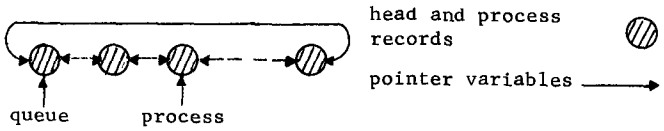

Fig. 2.Representation of a queue by a doubly-linked list

Each queue is headed by a dummy record which serves to identify the first and last processes (if any) in the queue. When the queue is empty, the head record is linked to itself. The head and process records are referenced through pointer variables.

It is a standard exercise to design Pascal procedures which initialize an empty queue, insert a record in a queue as the predecessor of another record already in the queue, delete a record from its present queue, and determine whether a queue is empty.

The process records in a queue can be enumerated in their order of priority as follows:

```
var queue, process: ↑process set;
begin process:= queue↑.succ;
      while process ≠ queue do
      process:= process↑.succ;
end
```

5. Processor Algorithm

At the level of description used here, a processor is capable of operating directly on process records and queues, and the synchronizing primitives wait, signal, start, and stop are considered machine instructions. Furthermore, we do not distinguish between central processors and peripheral devices. The latter are just processors dedicated to the execution of fixed processes. Synchronization of processes is controlled by semaphores only. It may be necessary, at the most primitive level of programming, to implement this abstraction by means of hardware registers and interrupts. Once this has been done, however, these technological tools become as irrelevant to the programmer as the logic circuits used to build an adder. The pitfalls of forcing programmers to think in terms of interrupts instead of semaphores have been aptly described by N. Wirth[6]. In fact, as several people have pointed out, there are good reasons to consider even semaphores as being too primitive and dangerous for general use. I leave it as an exercise for the reader to use Pascal to define a more appropriate synchronizing tool: a sequential file of messages in terms of standard data types and semaphores.

The instruction execution algorithm of one of the abstract processors is the following:

```
var process: ↑process set;
begin repeat
        initiate(process, ready queue);
        label idle
        begin with process↑ do
              repeat
                examine(process);
                execute instruction(program pointer);
                program pointer:= next(program pointer);
              until false;
        end
      until false;
end
```

When the processor is idle, it initiates the execution of the most urgent (i.e., first) process in the ready queue and continues to execute it until it is time to preempt it for one reason or another. Whether or not a running process should be preempted is decided inside the procedure examine. The processor refers to the description of its current process by means of a local variable process. (In Pascal, local variables are declared before the begin delimiter).

The structured statement

with r do S

where r is a record identifier, enables the statement S to refer directly to identifiers of the record fields without qualifying them with the record identifier, e.g., using "program pointer" instead of "process↑. program pointer". The meaning of the label idle will be explained later.

It is assumed that the computer system includes a sequential switching circuit, called an arbiter, to which all processors are connected. This circuit and two machine instructions, enter and leave section, are the hardware implementation of a single critical section

critical mutex do S

which ensures that the examination and modification of process records, performed by the concurrently operating processors, exclude each other in time. This critical section (which involves the busy form of waiting) will be used only at the short-term level of scheduling. On top of it, we construct the wait and signal primitives which enable processes to establish an arbitrary number of other critical sections (using the non-busy form of waiting).

I can now define the behavior of an idle processor which examines the ready queue repeatedly until it finds a process which can be initiated:

```
procedure initiate(victim, queue: ↑process set);
var busy: boolean;
begin busy:= false;
      with queue↑ do
      repeat
         critical mutex do
         if not empty(queue) then
         begin victim:= succ;
               remove(victim);
               victim↑.state:= running;
               busy:= true;
         end
       until busy;
end
```

When a processor must transfer a running process to a queue of waiting processes, it scans the queue and inserts the process according to its priority:

```
procedure delay(victim, queue: ↑process set);
var next: ↑process set;
begin with victim↑ do
      begin state:= waiting;
            next:= queue↑.succ;
            while priority ≥ next↑.priority do
            next:= next↑.succ;
      end
      insert(victim, next);
end
```

This algorithm assumes that small numbers denote high priority, and that the priority field of the head record is so large that it always terminates the search cycle.

6. Process Synchronization

The data type semaphore is a structure consisting of a counter and a queue of waiting processes:

```
type semaphore = record counter:  integer;
                        queue:  ↑process set;
                 end
```

The wait operation is defined as follows:

```
procedure wait(s:  semaphore);
begin critical mutex do
      with s do
      if counter > 0 then
      counter:= counter - 1 else
      begin delay(process, queue);
            exit idle;
      end
end
```

The exit statement, executed after the transfer of a running process to the semaphore queue, causes a jump to the end of the nearest enclosing compound statement headed by the label idle, that is, to the point in the instruction execution algorithm where the processor will initiate the execution of another ready process. This disciplined kind of jump is a restricted form of the goto statement defined in the Pascal report[5]. It was originally suggested by P. J. Landin[7] in a more general form which associates a procedure with the label.

The signal operation follows below:

```
procedure signal(s:  semaphore);
var victim: ↑process set;
begin critical mutex do
      with s do
      if empty(queue) then
      counter:= counter + 1 else
      begin victim:= queue↑.succ;
            remove(victim);
            delay(victim, ready queue);
            with victim↑ do
            program pointer:= next(program pointer);
      end
end
```

Notice, that when a process executes a wait operation the increase of its program pointer is delayed until the process is allowed to continue. This ensures that, if a process is stopped while it is waiting on a semaphore, and, later, started again, it will automatically repeat the incomplete wait operation.

7. Preemption and Resumption

Before a process is stopped, and possibly terminated, we must be very careful to allow it to complete the execution of all critical sections initiated by it. In general, therefore, a process cannot be stopped instantly. All one can do is to request that its processor transfer it to the stopped state as soon as possible. This problem is handled by extending each process record with a boolean and an integer, called the stop request and section depth, respectively. The latter is a count of the number of incomplete critical sections entered by the process. The reader will notice, that these variables serve the same purpose as the interrupt and inhibition bits at the hardware level. Finally, each process record is extended with a stop queue in which other processes can wait until the process in question has stopped.

Preemption by means of the stop primitive makes it useful to distinguish between semaphores used to synchronize critical sections and input/output relationships. When a process enters a critical section, its section depth must be increased by one to guarantee that it will be able to complete that section, and, when it leaves it again, the section depth must be decreased by one. However, if we consider a set of sender and receiver processes communicating through a common semaphore, it is quite possible to stop one of the receivers even though it may be waiting for a signal from a sender. Consequently, the depth counters should not be changed in this case. This is another motivation for choosing different notations for the two uses of semaphores.

Our abstract processor examines the description of its current process after each instruction in order to determine whether another process wants to stop it, or whether it should be preempted in favor of a more urgent process in the ready queue. Again, we notice in passing, that the processing time required for this evaluation can be reduced to tolerable proportions by the well-known technique of clock interrupts.

```
procedure examine(process: ↑process set);
begin critical mutex do
      with process↑ do
      if stop request & section depth = 0 then
      begin complete stop(process); exit idle end
      else
      if not empty(ready queue) then
      if priority > ready queue↑.succ↑.priority then
      begin delay(process, ready queue); exit idle end
end
```

The procedure complete stop is fairly trivial: it resets the stop request of a given process, changes state to stopped, and transfers all processes waiting in its stop queue to the ready queue.

The stop and start operations work as follows:

```
procedure stop(victim: ↑process set);
begin critical mutex do
      with victim↑ do
      if section depth = 0&state = waiting then
      complete stop(victim) else
      if state ≠ stopped then
      begin stop request:= true;
            delay(process, stop queue);
            exit idle;
      end
end

procedure start(victim: ↑process set; level: integer);
begin critical mutex do
      with victim↑ do
      if state = stopped then
      begin priority:= level;
            delay(victim, ready queue);
      end
end
```

8. Timing Constraints

It was pointed out in the introduction that the scheduling decisions taken at the short-term level determine the rate at which the computer system is able to respond to real-time events. As an example, consider a computer system in which storage words can be accessed in 1 μsec. An outline of the primitives wait, signal, start, and stop in typical machine code shows that they have roughly the same execution time. Assuming that an average of ten processes must be examined when a given process is transferred to a queue, I find that the proposed system can respond to internal and external signals within 100-300 μsec. This means that scheduling decisions taken by processes at the medium-term level will take at least, say 1 msec each. Thus, it is unrealistic to change priorities more frequently than, say, every 20 msec.

Although these figures are acceptable in most environments, there are certainly applications which require response to more than 3000-10000 events per second, e.g., speech recognition. The figures may, of course, be improved by implementing the primitives in hardware. Nevertheless, I believe it is a valid conclusion that decisions taken at the short-term level of scheduling by no means are innocent, and a realistic designer must be prepared also to change this part of the system.

Acknowledgments

I wish to thank Nico Habermann, Anita Jones, Alan Perlis, and Bill Wulf for constructive comments.

References

1. Brinch Hansen, P. The Nucleus of a Multiprogramming System. Comm. ACM 13, 4 (April 1970), 238-250.

2. Dijkstra, E. W. The Structure of THE Multiprogramming System. Comm. ACM 11, 5 (May 1968), 341-346.

3. Lampson, B. W. A Scheduling Philosophy for Multiprocessing Systems. Comm. ACM 11, 5 (May 1968) 347-360.

4. Saltzer, J. H. Traffic Control in a Multiplexed Computer System. MAC-TR-30, Massachusetts Institute of Technology, Cambridge, Mass., July 1966.

5. Wirth, N. The Programming Language Pascal, Acta Informatica 1, 1 (1971), 35-63.

6. Wirth, N. On Multiprogramming, Machine Coding, and Computer Organization. Comm. ACM 12, 9 (September 1969), 489-498.

7. Landin, P. J. A Correspondence Between Algol 60 and Church's Lambda Notation, Parts I-II. Comm. ACM 8, 2 (February 1965), 89-101, and 3 (March 1965), 158-165.