INSTITUT FÜR PLASMAPHYSIK

GARCHING BEI MÜNCHEN

Studies in Operating Systems Design
Part I:
Semaphore Operations and Task Control.

Friedrich Hertweck

IPP 6/63 Januar 1968

Die nachstehende Arbeit wurde im Rahmen des Vertrages zwischen dem Institut für Plasmaphysik GmbH und der Europäischen Atomgemeinschaft über die Zusammenarbeit auf dem Gebiete der Plasmaphysik durchgeführt.

IPP 6/63 F. Hertweck

Studies in Operating
Systems Design
Part I:
Semaphore Operations
and Task Control.

January 1968 (in English)

ABSTRACT:

Certain aspects of a supervisory system regarding efficient handling of I/O are discussed. After introduction of the concept of "semaphore operations" a few specific ones are defined and it is shown that they suffice to describe the concurrent operation of a main processor and I/O devices. By means of these operations, a general scheme is set up to organize the I/O data flow using buffers for intermediate storage.

1. INTRODUCTION 2

2. BUFFER COUPLED PRODUCER/CONSUMER PAIRS 3

3. DIJKSTRA'S SYNCHRONIZING PRIMITIVES 6

4. A "REAL" COMPUTER 8

5. SEMAPHORE OPERATIONS 10

6. THE QUEUING OF TASKS 13

7. PRODUCER/CONSUMER PAIRS WITH SEMAPHORE OPERATIONS 17

8. THE WAIT MECHANISM 20

9. ONCE MORE: THE PRODUCER/CONSUMER PAIR 26

10. CONCLUSION 28

1. Introduction

In the present report we try to develop some of the concepts necessary for describing an operating system. An operating system has many aspects, but certainly one of the most important is the throughput it can offer. The mode of operation for second generation computers was to use a satellite for primary input and final output. Most of the third generation computers are apparently devised in such a way that they should be able to support their own Input/Output. However, this requires a lot of sophistication in that part of the Supervisor programme, which queues and controls tasks for the I/O devices.

Viewed with respect to dataflow, a lot of the I/O activity can be described in terms of producers and consumers which fill and empty buffers. These ideas are described in Sect. 2. After a short review of the proposal for "synchronizing primitives" as given by Dijkstra (sect. 3) we turn to the "real computer" that we have to deal with. We confine ourselves to what we consider the basic property of the "semaphore operations" (sect. 5). Having introduced the mechanism for the queuing of tasks (sect. 6) and the wait mechanism (sect. 8) we are able to describe in a concise way producer/consumer pairs which work on a segmented buffer. In our presentation we have to be somewhat vague in describing certain features. A more detailed description of these aspects of the supervisor will be presented in a subsequent report by K.H. Goihl [2].

2. Buffer Coupled Producer/Consumer Pairs.

The producer/consumer pairs belong to the most important processes going on within a computer system. Their main purpose is to smooth out the differences in the data production and data consumption rates of the various I/O devices and the CPU. A typical example is shown in Fig. 1.

The normal computer system will have one CPU but several "data channels" and I/O-devices. For example, there may be several input readers or output writers. If the readers, for instance, are equivalent, then there is no need to treat them differently except that the data read are stored under separate labels on the disc. But all the readers feed the same input data region on the disc. Since the generalization to multiple input devices is fairly obvious (it can basically be achieved by reenterable coding for the processes) we shall confine ourselves to simple processes.

In Fig. 1, process P_1 will read cards into its associated input buffer. P_2 will move the data to a disc storage. When all the data belonging to a job are read in, the job name will be placed in the list of jobs waiting for execution. P_3 will collect the data from the disc and place it in sequential order into Buffer 1. Processes P_4, P_5 and P_6 act in a similar way in the reverse order. Essentially, the disc storage is a buffer and all the processes, including the main processor, can be considered as buffer-coupled producers/consumers. Apparently some of the processes, e.g. P_2, look like consumers at one end and as producers at the other end.

The basic idea for any operating system is to consider the processes P_1, ..., P_6 and the CPU programme as independent of each other. P_1 should read cards whenever the card reader

contains cards and a buffer is available to put the data in. P_2 should transport data from the input buffer to the disc whenever the buffer is full and the disc (including the channel) is available. P_3 will select data according to some rule (e.g. first-in, first-out or some priority scheme) and read it from the disc storage into Buffer 1.

The problem now is to devise the processes P in such a way that they can work independently of each other but nevertheless are synchronized as required in a safe and efficient way.

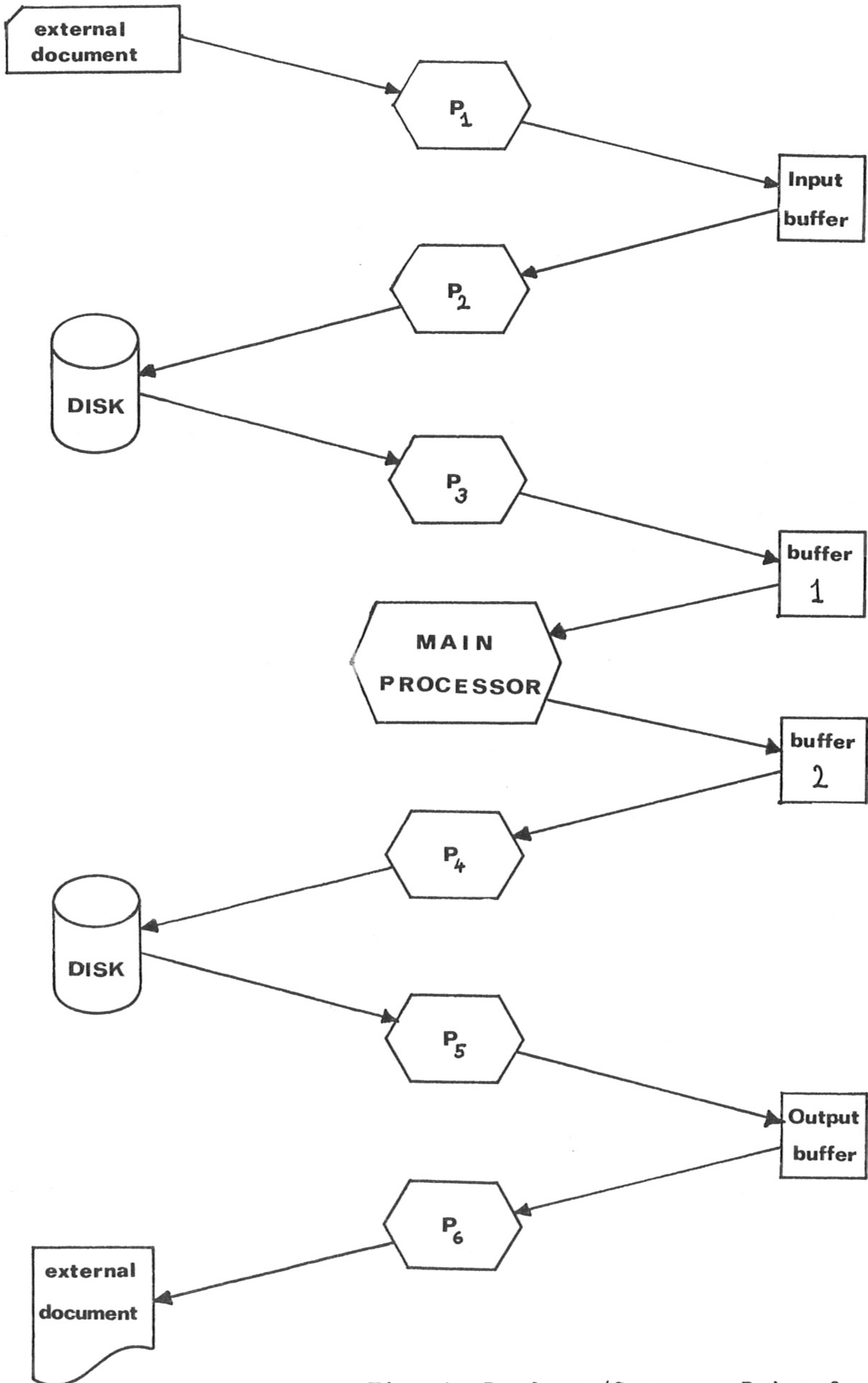


Fig. 1. Producer/Consumer Pairs for handling I/O streams during job processing.

3. Dijkstra's Synchronizing Primitives.

Dijkstra [1] has studied the cooperation of "loosely connected" sequential processes. According to his definition, loosely connected processes are "processes which can, apart from the rare moments of explicit intercommunication, be regarded as completely independent of each other". This applies in particular to their relative speeds. Processes communicate through common variables which are set and interrogated. The inspection or changing of such a common variable is considered an indivisible operation. He introduces the "semaphore variables" and two operations, the V-operation and the P-operation, which operate on a semaphore.

The definitions are:

The V-operation is an operation with one argument, which must be the identification of a semaphore. (If s denotes a semaphore, we can write V(s).) Its function is to increase the value of its argument semaphore by 1; this increase is to be regarded as an indivisible operation.

The P-operation is an operation with one argument, which must be the identification of a semaphore. (If s denotes a semaphore, we can write P(s).) Its function is to decrease the value of its argument semaphore by 1 as soon as the resulting value is non-negative. The completion of the P-operation - i.e. the decision that this is the appropriate moment to effectuate the decrease and the subsequent decrease itself - is to be regarded as an indivisible operation.

We may symbolically describe the semaphore operations by:

V(s) $:\equiv \{ s := s+1 \}$

P(s) $:\equiv P: \{ \underline{\text{if}}\ s > 0\ \underline{\text{then}}\ s := s-1\ \underline{\text{else}}\ \underline{\text{go}}\ \underline{\text{to}}\ P \}$

The brackets { } denote that the operations within are indivisible.

If we consider two processes A and B which are a producer/consumer pair we may write

s = 0;

Producer:	Consumer:
A: produce next portion;	B: P(s);
add portion to buffer;	take portion from buffer;
V(s);	process portion taken;
<u>go to</u> A;	<u>go to</u> B;

Initially, s = o. The processes A and B are then started. Process B will be stopped by P(s) which can be completed only after process A has performed V(s). Process A will produce the next portion and add it to the buffer. If A produces a burst of data, then s will be > 1 and B will work until s = o. This solution is acceptable if the two processes are performed in two independent processors. Since the processor which is to perform process B will wait most of its time within P(s), this is certainly a very uneconomical way to proceed.

4. A Real Computer

Though there are computer systems which contain a larger number of processors the normal structure of a system is an aggregate of a single processing unit (a "CPU") and a sufficiently large number of I/O channels which can operate independently of the CPU. However, the channels are not capable of completely maintaining the I/O operations since for starting and terminating these operations they have to be supported by programmes performed by the CPU. The termination of the I/O-operation within the channel will normally be indicated to the processing unit by an "interruption", i.e. the programme currently being performed is hold up and the (normally very short) programme for logically terminating the I/O operation is executed, whereupon the interrupted CPU programme is resumed.

If we turn to the producer/consumer pair we see that some parts of A and B have to be performed in the processing unit. This is especially true for the semaphore operations V(s) and P(s).

Before we go on, we impose some restrictions as to the nature of the processes involved. If A and B are two processes performed entirely by the processing unit there will be no parallel operation since we have only one processing unit and therefore the processes must be performed sequentially. If, however, the processes involve I/O operations, as is the case with producer/consumer pairs that transfer data from one secondary storage medium to another via the central store, these processes can go on independently if most of their time consists of I/O operations. In what follows we shall confine our discussion to those situations where a certain number of independent I/O processes is maintained by the CPU.

It is obvious that we have to look for a better solution for the P-operation. Neither should the solution lead to unnecessary tests for s while s is still zero, since this testing consumes CPU time, nor should there be, when s goes from zero to one, a long delay before the waiting process can continue.

5. Semaphore Operations

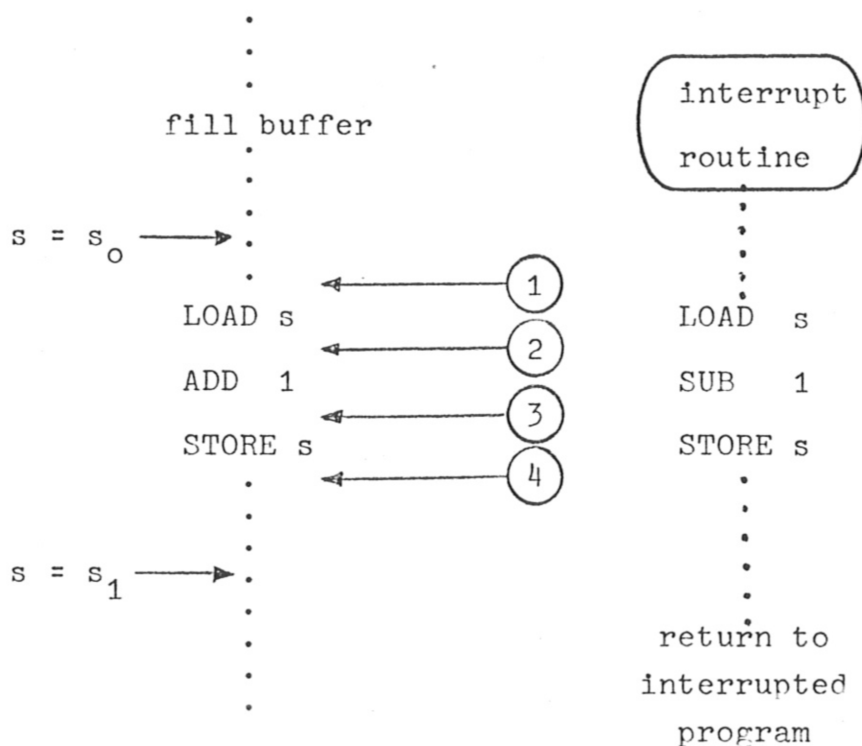
If we look at the definition of the P-operation (sect. 3) we see that an important property of it is that the value of its argument has to be decreased by 1 as soon as the resulting value is non-negative. This certainly is a desirable property since it ensures that the process, being held up by P, can go on as soon as the variable is changed to 1 by a V-operation. Hence we have to look for an equivalent of the P-operation which would still have this property but avoids any unnecessary looping.

In order to be more flexible, we shall assume that there can be more types of "semaphore operations" and not just the two operations introduced by Dijkstra. At first sight it might seem that we are complicating things unduly and that an operating system built up by using only two operations is more elegant. However, since we are dealing with the basic functions of any operating system our main concern - apart from complete security, of course - has to be efficiency.

We now make the definition:
There are semaphore operations implemented in the system which serve for the communication between various processes through the assignment and/or test of certain common variables. These semaphore operations are to be considered as indivisible operations.

It might appear - since we are dealing with a system with only one central processor - that there is no need for demanding "indivisible" operations, because all semaphore operations are performed by the central processor. We can show from the following simple example that this is not true. Let us assume that a CPU programme has filled a buffer segment and has called an output routine to write that segment onto tape. During the tape operation the

next segment of the buffer is being filled and this is completed just about the same time the tape operation has been terminated. The CPU programme will increase the segment count s at that time by 1 and the interrupt routine for the tape operation will decrease the segment count s by 1. The two programmes are basically



We assume that during the filling of the buffer $s = s_o$ and that after the filling it has to be set to $s_o + 1$. If, however, between these two points the interrupt of the tape unit is taken, the final result should be $s_1 = s_o$ since the interrupt routine will decrease s by 1 and therefore the two operations cancel out. The following table gives the possible results, depending on when the interrupt is taken:

case	s_1
1	s_o
2	$s_o + 1$
3	$s_o + 1$
4	s_o

If the sequence LOAD S / ADD 1 / STORE S were a semaphore operation, i.e. indivisible, then the interruption could take place only before the operation has started (case 1) or after the operation has been completed (case 4). Only then are we sure to get the correct result.

As we have shown, the need for the indivisibility of semaphore operations arises from the fact that the main processor can be interrupted. Hence, whenever we have to perform a semaphore operation in a "main programme" (i.e. a programme that basically uses CPU time and can be interrupted), we have to ensure that during the execution of the semaphore operation no interrupt can be taken. This requires the computer to have implemented appropriate instructions for enabling and disabling interrupts.

Semaphore operations, as used for the control of two coupled processes, generally occur in pairs and act on one or more common variables. Normally, one of the operations will be located in a main programme whereas the corresponding counterpart may be located in the interrupt routine. Since this latter routine will generally be uninterruptible we have only to take care of the semaphore operations located in main programmes. (We made these last remarks with the IBM System /360 in mind, but we feel that they may apply to other computer systems as well.)

So far we have not yet offered a counterpart to Dijkstra's V- and P-operations. This will be done in the next section.

6. The Queuing of Tasks.

Let us again consider the producer/consumer pair but this time taking a more realistic example:

s = o

Producer: A:	Consumer: B:
read data from tape;	P(s);
V(s);	write data onto disc;
go to A;	go to B:

We still use the V- and P-operations as we do not yet have a better mechanism. According to our stipulations in section 4 it is understood that these two processes "spend" most of their time waiting for the completion of the I/O operations.

It is obvious that these processes can go on concurrently only insofar as they are not using the CPU. If the CPU time for these processes is small compared with the data channel time which it normally is, then they can go on concurrently. We have tacitly assumed that the buffer space to read the tape is infinite. Then producer A can repeatedly read data from tape. The same is true for the consumer when writing onto disc. If we want several processes to go on concurrently, we have to provide for a queuing mechanism for all the processes which can go on.

We shall employ the two semaphore operations stack queue and select queue to achieve this. They can be defined as follows:

```
semaphore procedure   stack queue (processor, task, data);
     comment   p is equivalent to 'processor';
```

```
     begin
        put (task, data) into first empty position of queue (p);
        qcnt(p) = qcnt(p) + 1;
        if qcnt(p) = 1 then start task in first position of
                                                  queue (p);
     end;


semaphore procedure   select queue (processor);
     comment  p is equivalent to 'processor';
     begin
        remove task in first position of queue(p);
        qcnt(p) = qcnt(p) - 1;
        if qcnt(p) > 0 then start task in first position of
                                                  queue(p);
     end;
```

The procedure "stack queue" places a task which is ready to be performed into the queue for the specified processor. When a task is put into the queue, a check is made to see whether the queue was empty. If so, the corresponding processor is idle. Hence it is started with this task. If the queue contains already one or more entries, we may conclude that the processor is busy and only the stacking of the task has to be done.

The procedure "select queue" obviously has to be the last operation of a task, since it removes this task from the queue and selects a new one. (The task in the first position of the queue is the task currently being processed.)

Since these two procedures are semaphore procedures (stack queue is normally part of a main programme and select queue is located in the interrupt routine if the processor is a I/O device) the proper sequencing of tasks and the

maintenance of operation of the processors is ensured. To see this, we need only look at the case when the last task of the queue is being performed. If a "stack queue" is given before the "select queue", the number of tasks is increased from 1 to 2 and the subsequent "select queue" will remove the first task (now finished) and will start the new one. The number of tasks then remains 1. If the "select queue" comes first, the queue is emptied and the processor becomes idle. A subsequent "stack queue" finds the queue empty and therefore starts the processor with the task just stacked. From the semaphore property of the two operations it follows that only these two cases may arise but never a mixture. How this works is shown in Fig. 2.

The queuing mechanism may, of course, be expanded by taking into consideration priority rules for the selection of tasks. Then the selected task, if it is not the first of the queue, has to be exchanged with the first task of the queue. (see [2]).

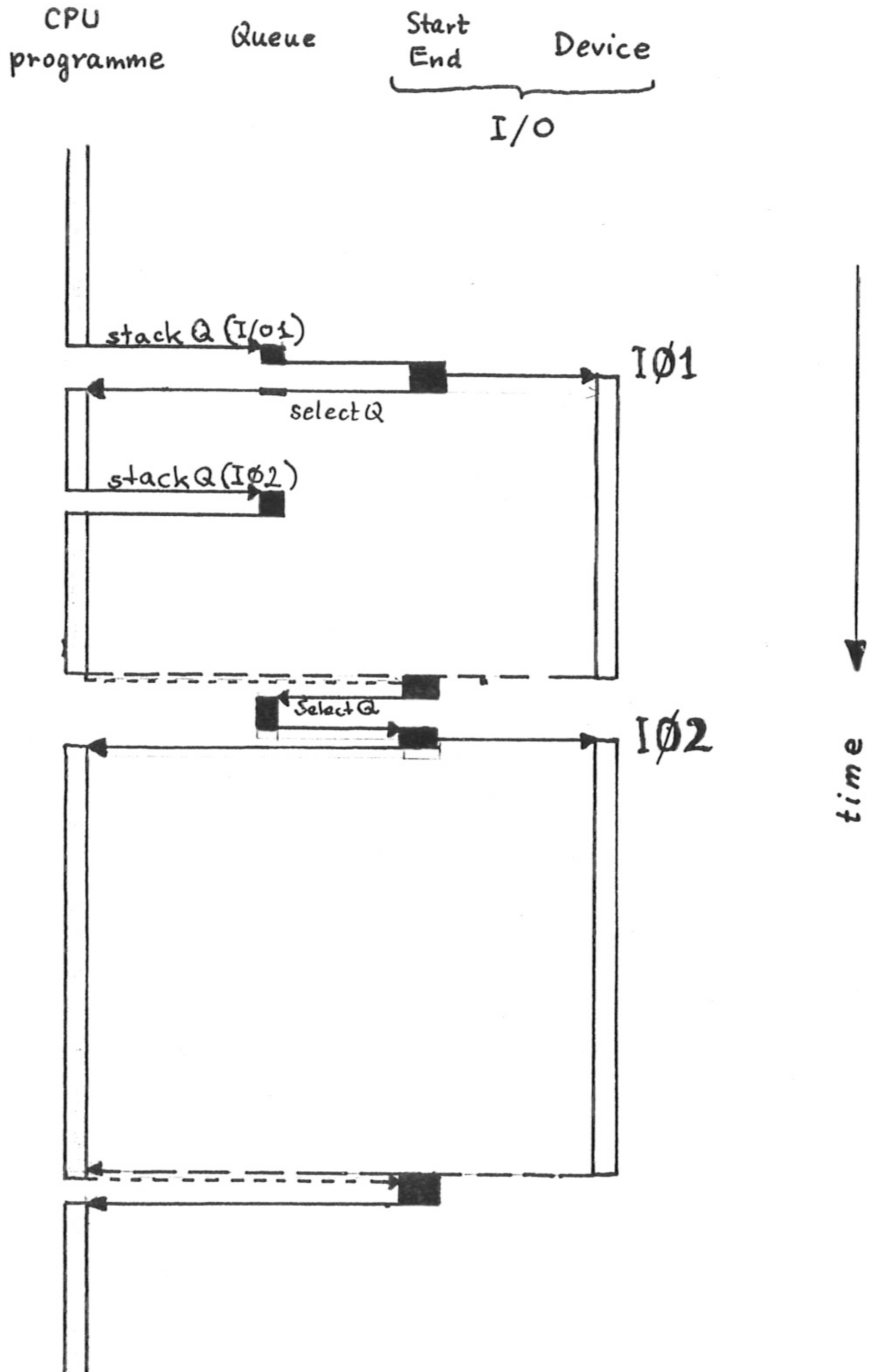


Fig. 2.

7. Producer/Consumer Pairs with Semaphore Operations.

We now define the following semaphore procedures:

(1) **semaphore procedure** S1 (s, processor, task, data);

```
    begin
            s := s + 1;
            if s = 1 then stack queue (processor, task, data);
    end;
```

(2) **semaphore procedure** S2 (s, processor, task, data);

```
    begin
            s := s - 1;
            if s ≥ 1 then stack queue (processor, task, data);
    end;
```

These two operations - being semaphore operations - are mutually exclusive.

Before we start to explain their functioning, we write down the example of the producer/consumer pair once more. Let us assume this time that the buffer is finite, divided into N segments arranged in a cyclic way. That is, if the last buffer segment is filled in one step, the next step will fill the first buffer segment. Associated with the buffer are three variables, q, np, and nc. q is the number of full buffer segments, np is the pointer to the segment which ist to be filled next and nc ist the pointer to the segment which is to be taken away next. The two processes are:

initial state: q := 0;   np := nc;

```
    Producer: P1:
    L: if q = N then go to L;
    read in buffer segment (np);
    np := (np + 1) mod (N);
S1: {q := q + 1;
     if q = 1 then stack queue (consumer, C1, buffer etc);}
    go to L;


    Consumer: C1:
    write out buffer segment (nc);
    nc := (nc + 1) mod (N);
S2: {q := q - 1;
     if q ≥ 1 then stack queue (consumer, C1, buffer etc);}
    select queue (this processor = consumer);
```

For the sake of clarity we have written down the two semaphore operations S1 and S2 explicitly. They are enclosed in brackets { } . As initial state, the buffer is empty (q = o, np = nc) and the two processes are idle. When the first buffer segment is filled, q is set = 1 and the consumer is started. For the time being we have written the producer as a cyclic programme with a wait loop if q = N. This only means that the producer is working at full speed as long as buffer space is available. We have used the same technique as for the stack queue/select queue pair. The consumer is started only when q goes from o to 1, otherwise the consumer stacks itself and restarts itself by the select queue operation. The pointers np and nc are only used in the producer and consumer programmes, respectively; therefore they need not be acted upon by semaphore operations. The only semaphore variable is q.

Taking q as the decision quantitiy whether to stack a task relieves the task queue of all superfluous entries and, at the same time, the scanning of the task queue in

case we use priorities is shorter. This is especially true in the case of data bursts in the producer, where the queue would have to accomodate N task entries instead of one. We have used, of course, the fact that these data are to be handled sequentially. The proper sequence of tasks is maintained by the sequence of buffers as described by q, np, and nc.

We have to improve the design of our producer/consumer pair in two respects. First, we have to abandon the dynamic wait loop when all buffers are full. This can be achieved by another semaphore variable indicating this overflow condition and by removing the producer from the task queue. The consumer has to check on the overflow condition and has to restart the producer after a buffer segment has become available. Secondly, the statements "read in buffer segment (np)" and "write out buffer segment (nc)" refer to I/O operations which are maintained by different processors (in this sense each channel or subchannel is a separate processor) whereas the "producer" or "consumer" are in fact CPU programmes.

The next problem we have to consider is therefore the request of services of one processor by another, and a mechanism which will make the requesting processor wait until the requested task has been finished. This will be done in the next section.

8. The Wait Mechanism

The purpose of the wait mechanism is to transfer a process to a temporary wait state until one or more secondary processes are completely finished. To describe this mechanism we use an algol-like notation like Dijkstra. The sequence of statements

$$\ldots\ S_A;\ \underline{\text{parbegin}}\ S_1; S_2;\ \ldots\ S_n\ \underline{\text{parend}};\ S_B;\ \ldots$$

is to be performed in the following way; First, statement S_A is completed. The statements S_1, S_2, ..., S_n are considered to be independent of each other and should be performed by n different processors in parallel. Statement S_B is started when and only when all of the processes within the parbegin/parend brackets are completed.

This notation is used by Dijkstra to describe algorithms which can be performed by parallel ideal processors. For our purposes we have to modify the meaning of the sequence of statements as given above. First of all, this sequence of statements always refers to a CPU programme. Within the brackets we may perform in parallel a number of secondary processes, i.e. I/O operations, and only one CPU programme. Moreover, the I/O operations have to be started by a CPU programme. Parallel operation is possible because the starting of an I/O operation requires a much shorter time than the I/O operation itself. In order to obtain parallel operation of I/O and the CPU S_n, the last "statement" within the brackets must be the CPU programme. The first $n - 1$ statements are the calls for I/O operations. Though they are performed sequentially, the I/O operations themselves are performed in an overlapped fashion.

It is clear from the bracket structure of parbegin/parend that they can be nested as in the following example (Fig. 3).

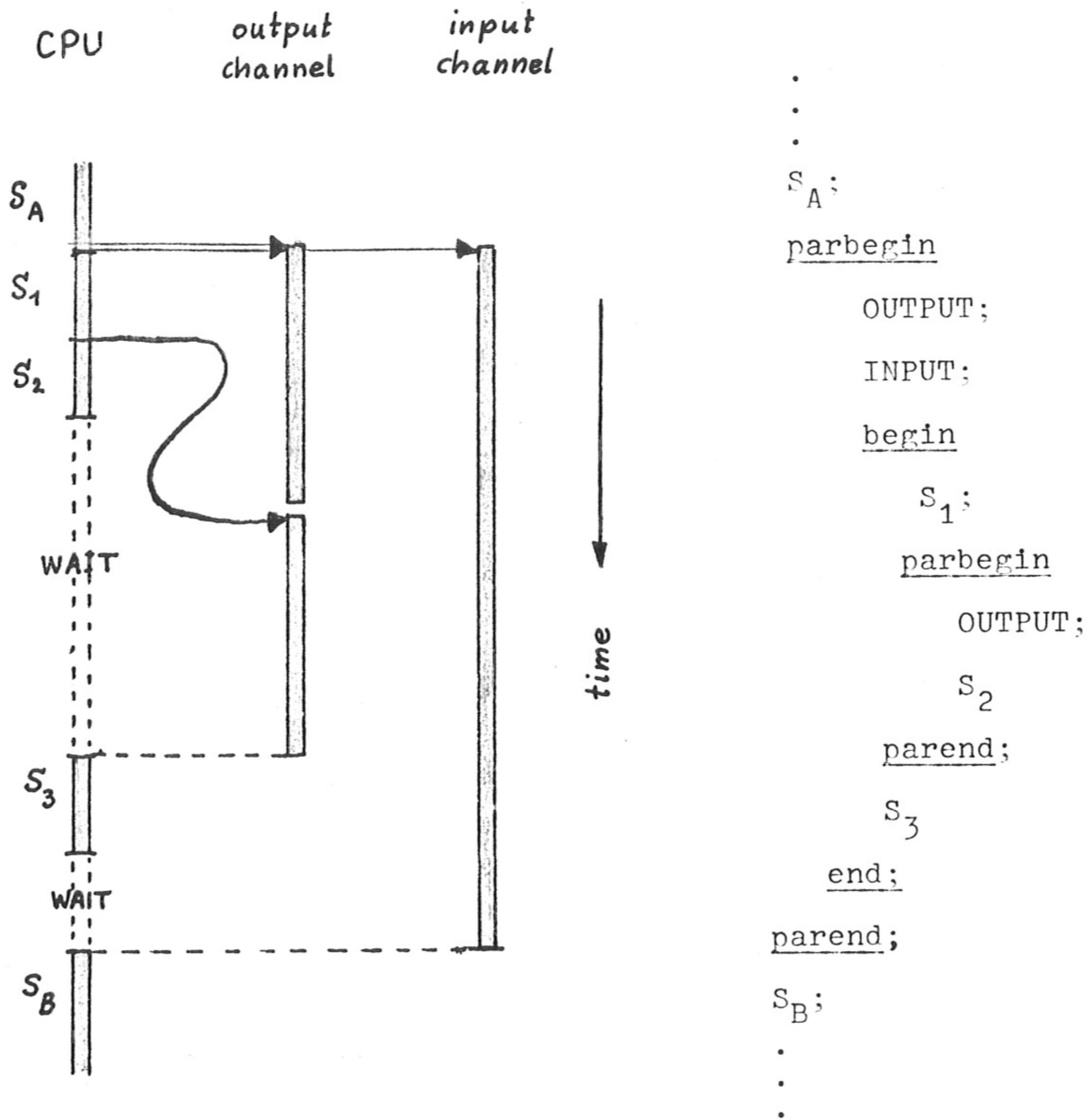


Fig. 3 . The wait mechanism.

So far the description of parallel operation has only been a formal one. We have to look for a way of implementation. We define the following semaphore operations:

```
(1) semaphore procedure parbegin (k);
        begin
            k := 1
        end;

(2) semaphore procedure par (k, processor, task, data);
        begin
            k := k + 1;
            stack queue (processor, task, data)
            comment data contain information about the
                main processor, the address of k, and the
                label of the continuation point (CONTINUE:)
                after 'parend';
        end;

(3) semaphore procedure parend (k);
        begin
            k := k - 1;
            if k > 0 then select queue (this processor)
        end;
        CONTINUE:

(4) semaphore procedure parexit
        begin
            k := k - 1;
            if k = 0 then
              stack queue (main processor, CONTINUE, data);
        end;
```

Let us now investigate whether these semaphore operations are capable of performing the required control of processes. Let us consider the simple case of one output operation and a conditional wait. We may write (for tape output, say):

```
    .
    .
    .
   parbegin (k);
   par (k, tape, write, data);
   parend (k);
   CONTINUE:
    .
    .
    .
    .
```

parbegin (k) sets the value of k to k = 1. The operation "par" starts the tape write operation and sets k = 2. It is important to remember that "par" has knowledge of the location of label "CONTINUE" and the nature of the main programme which contains the par-operation. This knowledge must be transferred to the secondary process - in this example the tape write operation.

When the operation parend (k) is performed two possibilities may arise:

(1) The tape operation is still in progress. Then k is reduced to 1 and the execution of the main programme is suspended. By the operation "select queue" within parend the main processor will be supplied with another task to work on.

(2) The tape operation has been terminated. Then parexit, which is performed at the end of the tape interrupt routine has reduced k to 1. The parend-operation in the main

programme will then reduce k to zero. The "select queue" is therefore skipped and the main programme will continue operation.

It remains to be shown that the parexit operation will restart the main programme in case parend has been performed first (thereby setting $k = 1$). Parexit will first reduce k to zero and then decide that this process (i.e. the tape write operation) was the last in the parbegin/parend bracket. It will therefore stack a main processor task into the queue. The entry point is at label "CONTINUE", and the information about the main programme, as received through "par", is used to reset registers, define priority, etc. The stack queue operation will restart the CPU with this task if it was idle at this time. If the CPU is busy with another task, this particular task is waiting in the queue.

The mechanism described works in a similar way if there are more secondary processes within the parbegin/parend brackets. Each "par" increases the value of k by 1 and each "parexit" decreases its value by 1. That process which resets k to zero is the last one which has been busy and will induce the continuation of the main task.

Fig. 4 shows an example where several tasks share the CPU.

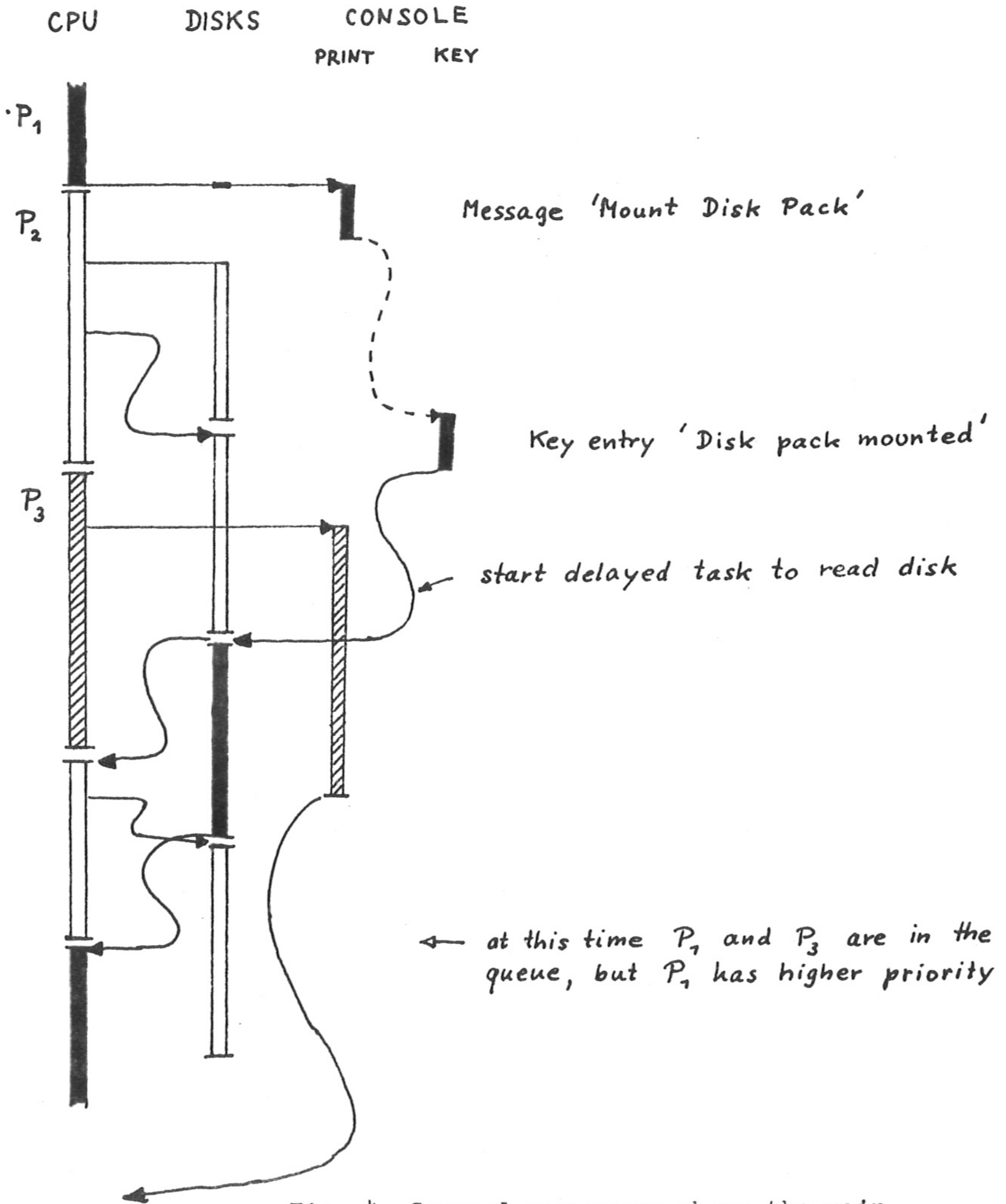


Fig. 4 Several processes share the main processor (CPU). Task switching is performed when waits for I/O are necessary.

9. Once more: the Producer/Consumer Pair.

In this section we shall present a final review of the producer/consumer pair making full use of the previously defined semaphore operations. We may write:

```
Producer:
{p = 0; if q = N then begin p = 1; select queue (CPU) end} ;
parbegin (k);
    par(k, tape, read tape to buffer, buffer segment (np));
parend (k);
np := (np + 1)mod(N);
{q := q + 1; if q = 1 then stack queue (CPU, Consumer, 0)} ;
stack queue (CPU, Producer, 0);
select queue (CPU);
```

The brackets { } denote the extent of the semaphore operations. We have assumed that the producer is started once from "outside". After that it cycles by placing itself repeatedly into the queue. The parbegin/parend pair ensures a wait for the data transfer. When the first buffer segment has been filled, the consumer is started within the second semaphore operation.

The consumer is described by

```
Consumer:
parbegin (k);
   par (k, disc, write buffer to disc, buffer segment (nc));
parend (k);
nc := (nc + 1) mod (N);
```

```
{ q := q - 1;
  if p = 1  then begin p := 0; stack queue (CPU, Producer, 0) end;
  if q > 0 then  stack queue (CPU, Consumer, 0) } ;
select queue (CPU);
```

The consumer starts by transferring a buffer segment to the disc. The completion of this operation is awaited by parend. When the buffer segment is empty, q is reduced by one If an overflow condition is present, then at this time the producer is restarted. The consumer restarts itself as long as q > 0, i.e. it has still something to do. If the consumer stops operation because the buffer is empty, the producer will restart it as soon as it has refilled one segment.

10. Conclusion

In order to describe some of the basic functions of an operating system we have introduced the concept of the "semaphore operation". In addition we have defined the special semaphore operations parbegin, par, parend, parexit for the control of wait conditions and the operations stack queue and select queue for the control of task sequencing.

It appears that these concepts are sufficient to describe such a system. For the time being, we have of course omitted a lot of important details. For instance, we have not said a thing about the structure of the routines for starting I/O operations for an I/O device or for handling interrupt conditions. We have only indicated in a rather vague fashion that at the end of an interrupt routine - which normally terminates the I/O operation logically - we have to place the parexit operation and after that the select queue operation to find out about another pending request for this device. Furthermore, we have not explained how the transport of data between tasks will actually be performed.

That these concepts are indeed sufficient to describe most of the I/O activity of a computer will be described in a subsequent report [2].

R e f e r e n c e s

[1] E.W. Dijkstra "Cooperating Sequential Processes" Technological University Eindhoven (1965)

[2] K.-H. Goihl "IPP Report" in preparation.