PDP-1 COMPUTER
ELECTRICAL ENGINEERING DEPARTMENT
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS
02139

PDP-35

INSTRUCTION MANUAL

PART 4 -- MULTIPROCESSING

9 November 1971

Introduction

As long as no IO operations take place, the "ethic" of a computer is to execute instructions, in the sequence specified by the programmer, as rapidly as possible. When IO operations are present, the computer usually must be slowed down to synchronize with the limited rate of information transfer of the IO devices. The simplest way to conceive of this is to think of the instructions that operate the IO devices as taking a long time to complete. For example, a type-out instruction keeps the computer busy for the entire length of time that the typewriter is operating. Only when the typewriter mechanism is finished with the character does the instruction complete and the computer go on to the next instruction. This method of doing IO is elegant because it makes the above-mentioned "ethic" still correct: execute instructions as rapidly as possible, in all cases. In most computers, this is one way of actually doing IO at the hardware level; in inexpensive computers, it is sometimes the only way.

To make absolutely clear what is meant, we list some IO operations:

Type-out (also paper tape punch, plotter, etc.) - When the instruction begins, the data is sent to the device. The instruction ends when the device has finished and is ready to be operated again.

Type-in - When a key is struck, the data is sent to the computer and the instruction completes.

Paper tape reader - Same as type-in, but the reader mechanism is started when the instruction begins.

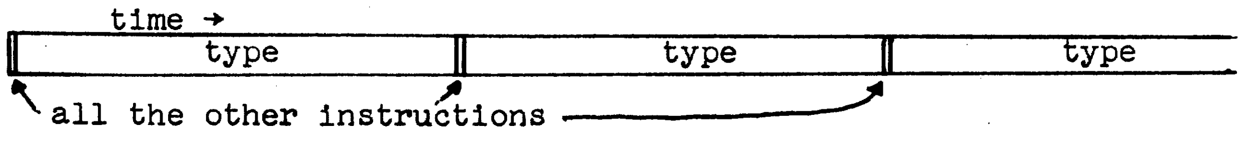
Drum or tape read - the operation begins when the instruction does. The instruction completes when the data are safely in core and the checksum has been verified.

Drum or tape write - The operation begins and ends when the instruction does

A program to type out characters stored in an array might look like this:

```
p,      lac array
        ivk 0     /type
        idx p
        jmp p
```

and a real time graph of its execution would look like this:



This makes horribly inefficient use of the processor, and in part this is an inevitable result of the fact that computers are fast and IO devices slow. However, there are ways to partially overcome this difficulty. Although there is nothing we can do to give us a 100 page listing in 2 seconds, the computer can be doing something else and not wasting its time.

1) In the case of output, if the data are moved to a buffer register as soon as the instruction begins, the processor doesn't need to be idle while the device is operating. It is only necessary that the computer, if it be allowed to continue, not initiate another instruction on the same device until the device is ready. This can be accomplished by making the second initiation hang up until the device is ready for it. The PDP-1 hardware behaves this way when output devices are operated in "pause" mode. (N.B. The preceding statement does not imply anything about the way the timesharing system operates; IO instructions are interpreted by the supervisor.) With this feature, a program that had a large amount of calculation between characters (say a real time printing of "e"), instead of behaving this way:

compute 1	type 1	compute 2	type 2	compute 3	type 3

during these times, the typewriter is idle

behaves this way:

	type 1		type 2		type 3
compute 1	compute 2	wait	compute 3	wait	compute 4

Similarly, if one didn't care about reliability, one could make a tape writing operation take up the data into a buffer and then complete immediately, allowing computation to proceed while the data are moved from the buffer to the tape. The PDP-1 timesharing supervisor does not do this, so that the user will be assured that, when the instruction completes, the transfer has taken place and, for example, the tape did not break.

This is a simple case of multiplexing the processor and an output device.

2) Something similar to 1) can be done for input. Suppose a compiler that runs about twice as fast as a tape reader is to make efficient use of the reader. It might operate like this:

read 1	read 2		read 3		
wait	use 1	wait	use 2	wait	use 3

This is a little harder to accomplish than the treatment of output devices, and generally requires two IO instructions: one to initiate the operation and another to wait for completion and obtain the data. The PDP-1 hardware does this. This is an extremely simple example of multiplexing the processor and an input device.

3) Another useful mode of operation is multiplexing two or more IO devices. For example, to obtain a listing of a file on a machine with two typewriters, the file could be divided in half and each half listed separately. Here a real synchronization problem appears, especially if the typewriters run at different speeds.

4) And, one might want to multiplex the processor with several IO devices. For example, while a batch job is running, the previous job's output could be listing on a line printer (having been stored on a high speed medium such as a disk) and the next job's input being read in and stored on a disk. Such a state of affairs, called SPOOLing (Simultaneous Peripheral Output On Line) is used in some of the less medieval batch processing systems. Here the synchronizing problem is even more formidable, especially since all three operations handle unrelated data, and the main program neither knows nor cares when a card has been read or when the printer is ready for another line.

5) The ultimate in multiplexing of unrelated operations is the timesharing system, which has not only to allow for complete generality in the mode of multiplexing of programs and IO devices, but has to allow the mode to change under program control.

There are three common ways of implementing a computer system that can do these things (particularly 3), 4), and 5)). One way is to use an interrupt (sequence break) system, in which IO instructions complete immediately without hanging up the processor. When the IO device completes, it interrupts the processor and starts it on a special "service" program. This program does whatever is necessary, e.g. obtains another line of output and sends it to the printer, and then resumes the original program. Another, much more expensive way is with a "data channel", which is a small processor, independent of the main processor, which executes the IO program (spending most of its time waiting) while the main processor runs at full speed on some other program. The third method is multiprocessing, in which the IO program is executed by what appears to be another main processor, identical to the original one. Because processors with the full instruction set are expensive, and the processor executing the IO program is idle most of the time, multiprocessing is usually (as in the PDP-1 timesharing system) "faked" with only one real processor which is multiplexed between programs.

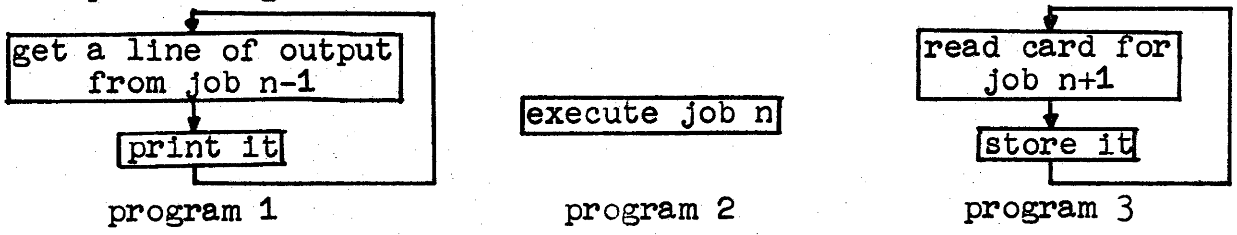
Multiprocessing is the most natural way of handling IO because:

It preserves the simple and elegant form of IO instructions given above, viz. an IO instruction hangs up the processor executing it until the operation is complete.

The IO programs are handled by a processor identical with the main processor, which is much more powerful than the typical data channel. This compatibility also simplifies communication between processors.

Multiprocessing is applicable to situations other than IO synchronization. In a system with more than one hardware processor, multiprocessing can be used to advantage in non-IO situations. Because the PDP-1 has only one processor, and the techniques of multiprocessing are the same in IO and non-IO cases, discussion will be limited here to IO applications. Even so, a subroutine can be considered as an IO device, and in such cases multiprocessing can sometimes bring a conceptual simplification (or a shorter program) which is worth the overhead of multiplexing a single processor.

Here is how application 4) (SPOOLing) would look when multiprocessing is used:



The hardware processor spends almost all of its time executing program 2.

Instructions

The "thing" that runs around in a program executing it is called a virtual processor or <u>process</u>. Each process has six "live registers": A, I, X, F (containing the six program flags and the address mode), G (the program counter, the overflow and extend mode bits), and W (a software register used for communicating with the supervisor). These registers are private to each process; no process can directly examine or modify the live registers of another process.

All processes running in the same sphere share the same contents of core memory and the same C-list.

The <u>frk</u> (fork) instruction (770042) creates a process. The frk is followed by the starting address for the new process. All live registers (except the program counter) are initialized to be the same as those of the process executing the frk. The process executing the frk continues two locations after the frk.

The <u>qit</u> (quit) instruction (770043) causes the process which executed the qit to be deleted. Unlike hlt and dsm, which cause the program's superior to take notice, a process executing a qit simply vanishes.

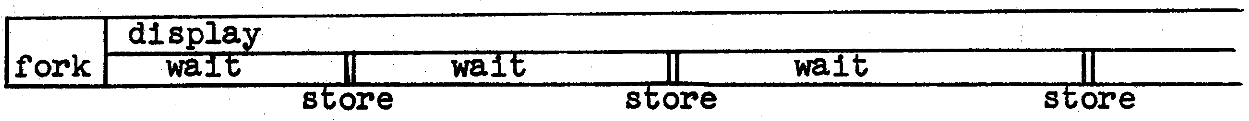
Example

Suppose it is desired to build an array of data and display the data while they are being added to the array. The data are being generated as a function of some input, e.g. type-in. It can be done this way:

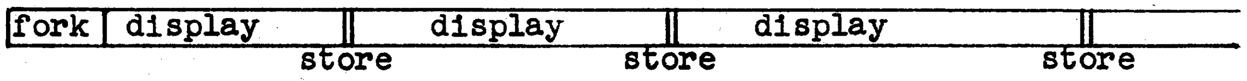
```
/start here with one process
        frk
        b
a,      ivk 200   /get input
        dac i p   /store it
        idx p
        jmp a

/second process will start here
b,      law m     /start at beginning of array
        dac q
c,      sad p     /reached end of array?
        jmp b     /yes
        lac i q   /no, get data
        jdp disp  /display data in desired format
        idx q
        jmp c
dimension q(1),m(1000)
p,      m         /pointer to end of array
```

Conceptually, a time graph of the execution of this program would look like:



Since the PDP-1 has only one real processor, the execution would actually look like:



Example of a Hazard or Lurking Bug

Multiprocessing programs are susceptible to a type of bug not found in ordinary programs, but similar to hazards and races in hardware. Suppose the first process in the above example wanted to decrease the size of the array:

```
law i 1   /(one's complement mode)
adm p
```

This would introduce a hazard. Suppose p contained m+101, q contained m+100, and the second process were somewhere in the display subroutine. p would be changed to m+100, q would be idx'ed to m+101, and the display would run off the end of the array, displaying garbage. Note that the bug will not be manifested unless p is decremented just at the time q is pointing at the end of the array. Hence, this program might be tested and found to operate correctly 100 times, which ordinarily would be fairly convincing evidence of its correctness. But it would still have a bug.

Hazards such as this are notoriously difficult to weed out in the normal process of debugging, because they are non-deterministic. Moreover, the odds of the hazard manifesting itself may be changed drastically by minor changes in parts of the program that have nothing to do with the real bug.

Because hazards may exist without immediately making themselves evident, those parts of multiprocessing programs concerned with process synchronization and control must be debugged on theoretical grounds rather than with trial runs. This memo contains many examples of properly written multiprocessing programs.

The scheduler is that part of the timesharing supervisor responsible for multiplexing the processor among those processes which can run. The process, and not the processor, is of concern to programmers. It is tempting to make assumptions about the likelihood of, say, one process executing a large number of instructions before another process has executed any, but to assume so is to invite hazards. The only guarantee the programmer has regarding the scheduler is the following: a process not hung in a wait will <u>eventually</u> execute its next instruction.

Process Synchronization and Control

Synchronization and control is a very important part of multiprocessing. Just as the stored program computer offers an immense variety of ways to realize a given algorithm, there are many ways to realize a solution to a control problem. Since core memory and the C-list are the only things that are shared between processes, all methods of synchronization use one of these two means.

One sequence that is very common in program control is called join:

```
        isp c
        qit
```

It joins n processes into one; the resultant process does not emerge until all n processes have entered. If the variable c is initialized to -n, the first n-1 processes that come to the join will quit, leaving -1 in c. The last process will skip over the qit.

In the above, c is a process control variable (sometimes called a semaphore). Process control variables are often used with the idx, adm, and isp instructions, for reasons that will become clear later.

Join presents another opportunity for a hazard. The initialization of the counter must take place before the first fork. If it is done this way:

```
        frk
        a
        law i 2   /(one's complement mode)
        dac c
        ...
        jmp j

a,      ...
j,      isp c
        qit
        ...
```

the process at "a" might do whatever it has to do and get all the way to the isp before the other process initializes c. Then, depending on what happened to be in c, no process would emerge, or one would emerge prematurely. While this may seem farfetched, it isn't. The effects of this bug may be easily demonstrated in practice.

Reentrant Programming

It is often desirable to have more than one process executing the same section of code, without any particular synchronization. At any instant, each process may be at any point in the section of code, independent of other processes. Such code is called reentrant or "pure".

Suppose that three typewriters are available, and a file is to be printed by being divided into thirds and each third printed. For simplicity, the file will be just an array. It could be done as follows:

```
/one process starts here
        nam
        frk
        g2
        frk
        g3
g1,     lac i p1
        ivk 1
        idx p1
        sas e1
        jmp g1
        jmp j

g2,     lac i p2
        ivk 2
        idx p2
        sas e2
        jmp g2
        jmp j

g3,     lac i p3
        ivk 3
        idx p3
        sas e3
        jmp g3
j,      isp c
        qit
        dsm
c,      -3
dimension a(60)  /contains the file
p1,     a
p2,     a+20
p3,     a+40
e1,     a+20
e2,     a+40
e3,     a+60
```

This is obviously wasteful of space, and should be done with one program executed reentrantly by each process. The processes will all use the same instruction sequence, but will operate on separate data. Since there are several things that must be private to each process, the index register will be used. This technique of using the index register to distinguish "private" areas of core is indispensible in reentrant programs.

```
/one process starts here
        iam
        ZX
        frk
        g
        SXXA
        sas (2
        jmp .-4
g,      aam
        lac i p
        xct i t
        idx i p
        sas i e
        jmp g
        isp c
        qit
        dsm
c,      -3
dimension a(60)  /contains the file
p,      a
        a+20
        a+40
e,      a+20
        a+40
        a+60
t,      ivk 1
        ivk 2
        ivk 3
```

Reentrant programs offer many opportunities for hazards. Suppose we want to print the square of each array element. If part of the program is written this way:

```
g,      aam
        lac i p
        dac w
        mul w
        scr 1s
        lai
        xct i t
```

a hazard will be introduced. Between the time that one process executes the dac w and the mul w, another process might execute the dac w. This particular bug would cause a malfunction very rarely, because when any one process gets to the dac w, the other two are almost certain to be hung on the xct i t. One correct way to write the above is to make w a private variable:

```
        dac i w
        mul i w
dimension w(3)
```

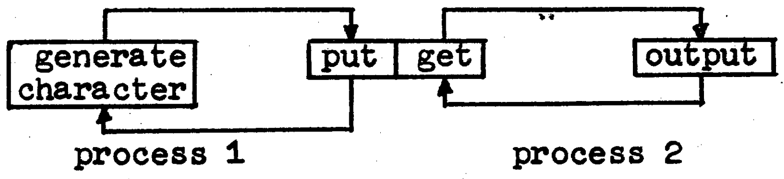
Another bug would be introduced if we tried to use a program-modified ivk instead of an xct:

```
        lio (ivk 1
        X+II
        dio .+1
        0            /ivk 1, 2, or 3
```

This sequence could be interfered with between the time the ivk is stored and the time it is executed. Program-modified instructions nearly always lead to hazards in reentrant programs. The jsp instruction is useful in reentrant programs, because jdp and jda lead to hazards.

Example

We now turn to a more involved example, a buffer. When used for output, this will "cushion" irregularities in the rate at which characters are produced, enabling the output device to run at full speed. One process takes characters produced somewhere and inserts them into the buffer. Another process takes the characters from the buffer and outputs them. Since the insertion takes very little time, the process generating characters doesn't have to wait, unless the buffer becomes full.



Process 1 sends data to process 2 through the buffer. Both processes must send control information to each other. More specifically, when the buffer becomes full, process 1 quits until space becomes available, and must be restarted (with a fork) when process 2 removes the next character. Similarly, when the buffer becomes empty process 2 must quit until process 1 inserts the next character. Let the capacity of the buffer be n and the variable p indicate its current level. The following will work:

```
/start here with one process
a,      [generate]
        [put]
        idx p
        sad (n
        qit
        sas (1
        jmp a
f,      frk
        a
b,      [get]
        [use]
        law i 1
        adm p
        sza i
        qit
        sas (n-1
        jmp b
        jmp f
p,      0
```

This works because the modification and test of p cannot be interrupted by the other process. If an interruption could take place between the modification and test (for example, if the test were done by

```
        idx p
        law n
        sad p
```

), or within the modification itself (for example,

```
        lac p
        sub (1
        dac p
```

), there would be a hazard. For example, if p were idx'ed from 1 to 2 by process 1, then reduced from 2 to 1 and tested by process 2, process 2 would not quit. When process 1 then finds that p contains 1, it will fork, thinking that p had previously been zero and hence process 2 had quit.

The idx, isp, and adm instructions have the required properties. An instance of one of these instruction is not interruptible, and, of two instances, one will always precede the other. The modification effected by these instructions appears to take place in a unique instant of time. Furthermore, they leave the result in A, so another instruction to load the result is unnecessary. The necessity of modification and test of a control flag without interference arises repeatedly in process control applications, making these instructions very useful.

Note that, if a standard ring buffer is used, with an input pointer and output pointer, "put" and "get" use only their own pointer and do not even look at each other's pointer, which will guarantee that the pointers are private and no hazards are associated with them. The fullness and emptyness checking is done entirely through the variable p, not by comparison of the pointers (the usual method of testing in ring buffers).

After checking that the generate, put, get, and use routines work properly, it is necessary to prove that the multiprocessing is correct. In this case, it must be shown that

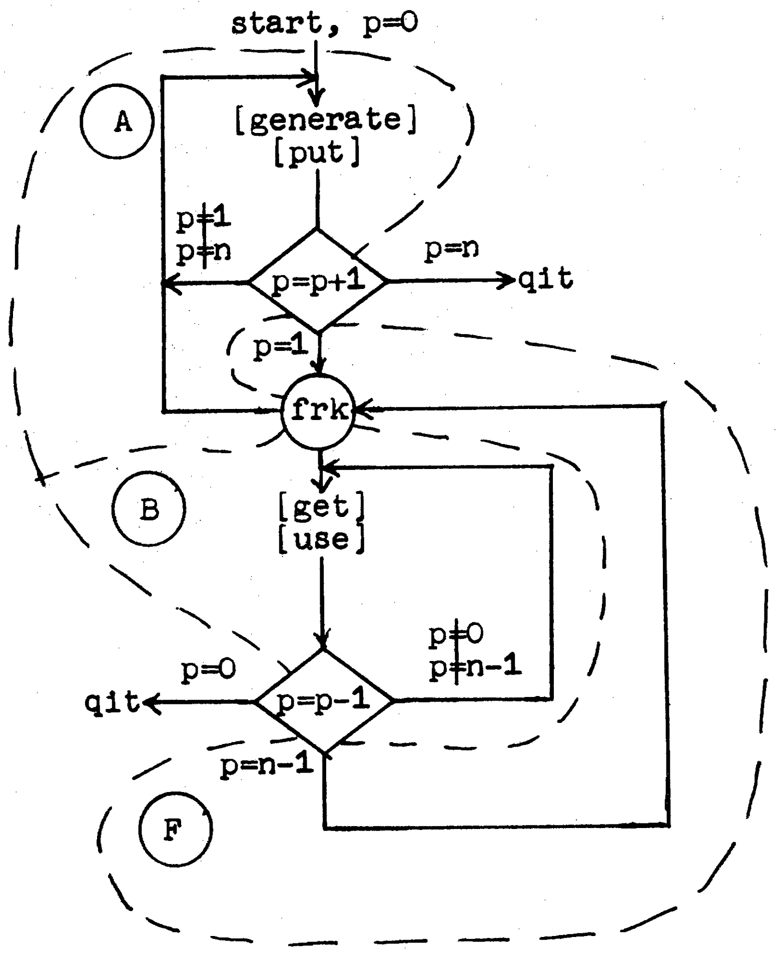
(1) The number of puts will not exceed the number of gets+n (the buffer will not overflow).

(2) The number of gets will not exceed the number of puts (the buffer will not underflow).

This is achieved by making sure that no process will run in a section of theprogram that will violate the above rules, i.e. no process will be in the generate/put section if the buffer is full, and none will be in the get/use section if the buffer is empty. Furthermore, it must be shown that

(3) No process will be unnecessarily deleted, i.e., the generate/put process will be removed only if the buffer is full, and the get/use process will be removed only if the buffer is empty.

The usual method of proving such a thing is to divide the flow chart into regions, such that each transition between regions is simultaneous with a modification of a semaphore, and then correlate the numbers of processes in the various regions with the semaphore states.



The only possible states are:

p	no. in A	no. in B	no. in F	
0	1	0	0	(start)
1	0	0	1	
1	1	1	0	
2	1	1	0	
...	1	1	0	
n-2	1	1	0	
n-1	1	1	0	
n-1	0	0	1	
n	0	1	0	

Exercise: Show that no other states can be reached, by considering the effect of any instruction execution on each state. Conclude that, since "get" is not executed when p=0 and p is increased to 1 after "put", the buffer will not underflow. Conclude similarly that the other two requirements are met.

Locks

Although, as has been seen, the instantaneity of idx, isp, and adm make it unnecessary in many applications, the ability to lock sections of a program is very useful. A lock guarantees that no more than one process will be in the locked region at any one time. For example, the reentrant program that needs a temporary can be fixed with a lock:

```
[lock]
dac w
mul w
[unlock]
```

If every variable or data object is either referred to only from inside of locked regions or referred to only from outside of locked regions, then all locked regions can be considered to be instantaneous transitions, with no hazards associated with them.

Ideally, lock and unlock should be considered to be addressable instructions, addressing a lock indicator. The lock operation guarantees that no more than one process at a time will be in any of the regions locked by the same indicator. The operation could be modeled by a lock indicator containing 0 or 1, initially 0. Lock waits until the indicator is zero, then sets it to one and proceeds. Unlock sets it to zero.

The PDP-1 timesharing system has hardware lock and unlock instructions (<u>lok</u> (770040), <u>ulk</u> (770041)) which have a few unusual properties:

They are not addressable. (All lok and ulk instructions could be considered to address the same global lock indicator.)

When a process is locked, not only are other processes prevented from entering this or any other locked region, but they are prevented from executing any instructions at all. During a lock, interrupts are disabled, giving the process the undivided attention of the computer.

Because of the potentially dangerous results of disabling interrupts, only 63 memory cycles (mul and div count for about 5 extra memory cycles) are permitted under a lock (including the lok itself). If this is exceeded, the "lock fault" trap occurs (ID prints "ↄↄ").

An instruction trap (a supervisor call such as mta's, ivk's, frk, qit, etc.) of any kind, or an address snag (an attempt to reference a core which has not been swapped in, even if the reference does not result in a address violation), will automatically remove the lock.

Lok and ulk are very fast, and are suitable to the application shown above.

The instruction sfa (770045) "skip if field assigned" will skip when the address contained in A(3-17) may be referenced without causing an addressing trap (note that a core may be "assigned" but swapped out — referencing such a core will cause an addressing trap which will inform the executive that it should be swapped in)

would someone please document the sfa (770045) instruction

Long Locks

It is sometimes desirable to lock a long region of code, usually in connection with resource allocation. For example, the microtape supervisor is reentrant, with one process for each transport. Only one tape unit can be actually transferring data at one time, so, when the desired block is reached, a lock is performed. The locked region contains many supervisor calls, and the lock may remain in effect for a large fraction of a second.

A lock can be implemented in software using isp and adm. Let w be the lock indicator, containing -2 if unlocked and -1 (or more) if locked. Lock will be:

```
isp w
jmp .+4
law i 1    /(one's complement mode)
adm w
jmp .-4
```

and unlock:

```
law i 1
adm w
```

If w contains -1, another process trying to enter the locked region will hang up in the loop. The law i 1, adm w undoes the effect of the isp. If it is necessary for the lock to hold two or more processes in abeyance at the same time, this has a hazard. If one process is in the locked region and two are trying to lock, w takes on the values -1, 0, or +1. The phase of the processes trying to lock might be such that w happens never to contain -1. (Such a state of affairs is unlikely to continue forever, but in principle it might.) After the unlock is executed, w might never contain -2, although one of the processes is entitled to go through. This can be fixed by changing the lock to:

```
law i 2
sas w
jmp .-1
isp w
jmp .+4
law i 1    /this is almost never executed
adm w
jmp .-7
```

This type of lock is addressable (the lock indicator, or semaphore, is location w), can last for more than 63 cycles, and is impervious to traps, but it is very inefficient because a process attempting to enter the lock burns up processor time while it waits. Furthermore, it has the following problem. Of several processes trying to enter, one will always succeed, but which one is a matter of chance. If processes arrive at the lock as often as processes arrive at the unlock, there will always be processes waiting to enter. It may happen that a particular process is unlucky, and never manages to get in.

During a long lock it is desirable that processes which are unsuccessfully attempting to enter not waste processor time. One solution is to have such a process quit after leaving some indication that it should be recreated with a fork as part of the unlock operation. Lock:

```
        isp w
        jmp l
        qit
l,      ...
```

Unlock:

```
        law i 1
        adm w
        SA>P
        jmp .+3
        frk
        l
```

Such a procedure is unsuitable in most applications because when a process unsuccessfully tries to lock, its live registers are lost. The process that eventually appears at "l" is a different process from the one that tried to enter. But if two or more processes are attempting to enter the locked region, they presumably intend to do different things. This intent is usually carried in live registers, which must be preserved somewhere. A table of preserved registers is needed, which requires that the number of processes be known to be bounded. In almost every case it is quite cumbersome. A shorter, faster, easier, and more elegant way is with a programmed queue.

Programmed Queues

A <u>queue</u> consists of a number called the population (which may be positive, negative, or zero), and a set of processes that are suspended as if in IO waits. When the population is zero or negative, no processes are suspended. The principle operations are:

mta 303

Create queue. The capability index is specified by the low 6 bits of A (zero specifies the first free index). The index is returned in A. The initial population is minus the → absolute value of the → contents of I. Skip if successful. Absolute value is done as a 1's mode computation. To avoid confusion, always specify I ≥0.

For queue ivks, the variant (bits 8-11 of the ivk instruction) normally specifies the operation. If the variant is zero, A(13-14)+1 is used.

Variant	Operation
1	Enter queue. The population is increased by one. If the result is strictly positive, the process is suspended in the queue. If the result is zero or negative, the instruction completes and the process continues.
2	Release queue. The population is decreased by one. If the population had been strictly positive, the process that has been suspended longest is removed from the queue and restarted as if its enter queue had just completed. The release queue instruction always completes immediately.
3	Read queue population to A (in one's complement). Intended mainly for debugging. Slow. (A faster way of determining the queue population is to maintain a variable which is incremented when the queue is entered and decremented when it is released.)

All live registers are preserved during all queue ivks, including an enter queue that hangs up. All queue ivks are uninterruptible, and appear to take place in an instant. When a programmed queue is used, the queue population is a semaphore. (In Dijkstra's terminology, enter queue corresponds to the P operation, and release queue to V.)

An efficient "long lock" is trivial with programmed queues. Use a queue with an initial population of -1.

```
        ivk 100+queue
        ...       /locked region
        ivk 200+queue
```

The first process to enter will set the population to zero, after which all other processes will hang up until the first emerges.

There is a variation of a lock in which the number of processes in the locked region must be limited to some fixed number N, not necessarily 1. This is useful in reentrant subroutines that need to allocate buffer and variables space when they are called, and have a limited capacity to do so. (In applications like this, the preservation of live registers through the lock is critical.) With a programmed queue, this kind of lock is as easy as the ordinary kind. The initial queue population is simply set to -N instead of -1.

As a final example of the power and simplicity of queues, the buffer problem is rewritten using two of them:

```
/start here
        law 1
        lio (n
        mta 303
        bpt
        law 2
        cli
        mta 303
        bpt
        frk
        b
/end of initialization
a,      ivk 101
        [generate]
        [put]
        ivk 202
        jmp a

b,      ivk 102
        [get]
        [use]
        ivk 201
        jmp b
```

The proof of the correctness of this program is left as an exercise for the now benighted reader.

Process Hoard

When a process executes a frk, the supervisor allocates a process. If no process is available, the frk waits until a process becomes available. A program may, if it wishes, insure that a process will be available when it is needed by setting its <u>process hoard</u>. The process hoard is the number of processes which the program is guaranteed to be able to have. It may have more than this number if, as is usually the case, the supervisor can allocate space for them. The hoard is simply the number that are reserved and guaranteed to be available at all times, no matter what other programs do. If the hoard is <u>n</u> and there are <u>m</u> processes currently in existence in a program, the next n-m frk's are guaranteed to succeed immediately. (The hoard is also of significance for entries. See Part 5 of this Instruction Manual.) The hoard should be set at the smallest value necessary to insure that every frk will complete eventually. For example, the "tree" program in the appendix may have up to nine processes running in it at once, but a hoard of four is sufficient. Upon logging in, the hoard is initially 1. Normally, the hoard is left at 1, since processes are nearly always available.

mta 406

Read process hoard to A.

mta 407

Set process hoard from A. Skip if successful.

Appendix

Here is a reentrant program which graphically demonstrates the meaning of a "fork".

```
/tree

beg,    lio (3
        law i 9.
        dac c
        lxr (374761
e,      ril 2s
f,      lpf
lup,    rcr 9s
        ral 9s
        iot 207
        rar 9s
        rcl 9s
        szf 2
        sub (2000
        szf 5
        add (2000
        szf i 3
        szf 4
        add (2
        lio (4000
        XMIXI<M
        jmp lup
        SII
        ril 7s
        TIXP
        jmp .+4
        isp c
        qit
        jmp beg
        rpf
        frk
        f
        rir 1s
        frk
        f
        jmp e

constants
variable
start beg
```