Published: 11/12/68

Identification

Overview of the Locking Strategy in the File System
A. Bensoussan

Purpose

Sections BG.19.00 to .03 describe the use of multiple locks in the Multics file system and the way they have been implemented.

BG.19.00 defines the problem and provides an outline of strategy.
BG.19.01 describes the locking strategy in page control.
BG.19.02 describes the locking strategy in segment control.
BG.19.03 describes the locking strategy in directory control.

Multiple Locks in Multics

We are referring here to the locks used to synchronize processes executing in ring zero. It would be easy to eliminate the problem by a single lock approach, allowing only one process at a time in ring zero. But we think it is worth having parallel processing in ring zero; therefore, we have to synchronize processes each time they need to share a ring zero datum, the value of which datum is subject to modification.

We have chosen to control parallel processing when each of the following set of data is involved:

- A directory
- A branch in a directory
- The hash table of the AST
- An AST entry
- The AST removal list
- The SST free storage area
- A PST entry
- All data needed to handle a page fault (except data needed for threading an AST entry in the removal list).
- APT

The way each of these data is protected by a lock associated with it will be explained in detail in the appropriate section. The observation we want to make at this point is the following:

When a process has locked a datum, say D1, and is now waiting for another datum to be unlocked, say D2, which has been locked by a different process, how can we be sure that the process that locked D2 is not waiting for D1 to be unlocked. This situation has been termed a "deadly embrace".

It is clear that if this situation occurred, both processes would wait for each other forever.

We must establish rules of behavior such that, if followed by all processes, the situation described above cannot occur.

We have tried to analyze the general problem, first, in order to find out what the basic requirements are; then we have treated the real problem in Multics as a special application, with its special characteristics.

<u>The General Problem</u>

The general problem can be defined as follows:

Several processes are sharing n different data; each process may need an unknown number of these data in an unknown order. How can we prevent the deadly embrace?

One solution would be for a process, upon finding required datum X locked, to take the following action: Restore all data locked by this process to the values they had before being locked, unlock them all, wait for datum X to be unlocked and restart at the point where the first datum was needed.

This can be described by algorithm 1 (also represented in Figure 1(a)).

Algorithm 1:

1. RESTART = Label of the instruction that tried to lock datum X.

2. If no data are locked by the process, then go to 9.

3. Null operation (Step 3 will be used later).

4. Y = Last datum locked by the process.

5. RESTART = Label of the instruction that tried to lock datum Y.

6. Restore all data to the values they had before Y was locked. (This step will be termed the "decompute operation".)

7. Unlock Y.

8. Go to 2.

9. Wait for data X to be unlocked.

10. Go to RESTART.

The operation described by Step 1 to 8, which has to be executed in order to be allowed to wait for a datum locked, will be referred to as the "back-up" operation.

The "decompute" operation is not easy to provide and we will try to remove the need for it as much as possible.

If we knew that, for performing a function, we need to lock only data A, B and C, we could start to alter any of them only after all of them have been locked. This way, if a process locks A and B and finds C locked, the back-up can be done without having to restore A and B since they have not been modified. When A, B and C have been locked, we can start to modify any of them because we know that no back-up will be needed from this point on.

But sometimes, and this is the case in Multics, it is not possible to decide that, from a certain point on, no other data will be needed; the reason is that, at any instant, the course of a process can be diverted to a temporary but unexpected path, for servicing a standard fault (for example segment fault or page fault in Multics), or for answering an interrupt, and a new datum may be needed in this path. We will call this path a "secondary path".

Our main objective is to make these randomly inserted secondary paths transparent to the main path as far as the need for backing up is concerned; that is, a back-up operation initiated in a secondary path should not have to unlock data locked in the main path.

Wait Hierarchy Organization

We present the concept of a wait hierarchy between data using a simple example first; then we describe it in its generality.

Let us assume that the number of data is only 2; the data are A and B. Instead of using the previous algorithm for preventing a deadly embrace, we can use the following one: "When a process has locked A and needs to wait for B, it can do so without unlocking A; on the other hand when a process has locked B and needs to wait for A, it has to back up as described in Figure 1(a)". We will say that data A has "wait permit" for B.

For the general problem in which we have n data, we must have a way of declaring that datum i has or has not "wait permit" for j, for any permutation of these n data taken 2 at a time.

This can be done by defining a boolean function W(i,j), that we call "wait permit function", which has a value for any i and any j (except i=j). By definition, datum i has wait permit for j if, and only if, W(i,j)=1. The notation $i \rightarrow j$ is equivalent to W(i,j)=1 and means that a process can lock datum i and wait for datum j without having to unlock datum i. The wait permit function must have the following properties:

1. $W(i,j) \wedge W(j,i) = 0$
2. $W(i,j)=1 \wedge W(j,k)=1$ implies $W(i,k)=1$

An example of such a hierarchical organization can be obtained by associating a unique number p(k) with each datum k and by defining the boolean function W as follows: $W(i,j) = p(i) > p(j)$.

If a wait hierarchy is established, a solution to the deadly embrace would be for a process, upon finding required datum X locked, to take the following action:

Back up until all data locked by this process has the wait permit for X.

This can be described by algorithm 2 (also represented in Figure 1(b)).

Algorithm 2:

Same as algorithm 1, except for step 3 where the null operation is replaced by:

3. If all data locked by the process has wait permit for x, then go to 9.

It is clear that, if all processes use this algorithm, no mutual blocking can occur: If a process, P1, is waiting for datum y with other data locked on its behalf, then any one of these data, say x, is such that $W(x,y)=1$. The process P2, that locked y, cannot wait for x with y locked since $W(y,x)=0$, by virtue of function W's property 1. Furthermore, if P2 is waiting for datum $z \neq x$ with y locked, this implies that $W(y,z)=1$; the process that locked z cannot wait for x with z locked since $W(x,z)=1$ by virtue of function W's property 2, and thus $W(z,x)=0$.

$$\begin{tikzcd} \boxed{x} \arrow[r] & \boxed{y} \arrow[r] & \boxed{z} \end{tikzcd}$$

The algorithm represented by Figure 1(b) has an advantage over the one represented by Figure 1(a): The back up operation does not necessarily require the unlocking of all data locked by the process. We will use this property in order to make a main path and a secondary path "back up independent".

A secondary path can be made transparent to the main path with respect to the back up, if we can define a wait hierarchy between data such that all data needed in the main path has the wait permit for all data needed in the secondary path.

General Method

The general method that we will use to solve the deadly embrace problem for a system in which secondary paths may be taken by a process is as follows:

1. Identify all data subject to locking.

2. Identify the different domains of execution in which main path and secondary paths may be executed.

3. Identify the data needed in each domain.

4. Establish a wait hierarchy by defining a wait permit function W(i,j) which has a value for any pair of data (except i=j) and has the two properties mentioned above.

5. Establish system restriction which inhibits the occurrence of any secondary path that might need to unlock data locked in the main path in order to satisfy the requirements of algorithm 2. In step 4, the wait hierarchy should be defined in such a way that it minimizes the system restrictions.

Multics Characteristics

1. List of data

   The list of all data which may be locked after a process has entered the file system is given below, with the nomenclature used to identify the locks associated with these data.

Lock Variable	Data Description
DIR(n)	Directory whose tree name is n.
BR(n)	Branch for segment whose tree name is n.
AST.HT	Hash Table of the AST.
ASTE(n)	AST entry for segment whose tree name is n.
PSTE(x)	PST entry for process whose id is x.
AST.RL	List of AST entries candidate for removal.
SST.FS	Free storage of the SST.
PC	All data needed to handle a page fault (except data needed to thread an AST entry in the removal list). The lock PC is used to enforce sequential processing in page control, core control and device control.
APT	Active process table. The lock APT is used to enforce sequential processing in the process exchange.

2. Domains of Execution

We can identify four domains in which a Multics process can be executing with a file system data base locked on its behalf; they are:

D(0).	Traffic Controller (TC).
D(1).	Page Control (PC).
D(2).	Segment Control (SC).
D(3).	Directory Control (DC).

A Multics process executing in a domain D(i) can be diverted unexpectedly from its normal course and be forced temporarily to executed a secondary path in domain D(j), with $i > j$ if $i \neq 2$ and $i \geq j$ if $i = 2$ (see Figure 2).

3. Data needed in each domain

The system requirements are such that in each domain a process needs certain data as shown below (and in Figure 3).

DC	SC	PC	TC
DIR	DIR		
BR	BR		
	ASTE		
	PSTE		
	SST.FS		
	AST.RL	AST.RL	
		PC	
			APT

4. The Multics Wait Hierarchy

Since the same data may be needed in the main path and in the secondary path, it is not possible to find a wait hierarchy such that a secondary path is always invisible to the main path as far as the back up is concerned; therefore, we will have to establish system restrictions. However, the wait hierarchy has been defined in such a way that a minimum of system restrictions will be needed. The justification for this hierarchy organization requires more background than we have at this point, and will be given in the appropriate MSPM BG.19 section.

The notation $a \rightarrow b$ is used for: data "a" has the wait permit for data "b", that is $W(a,b)=1$ and $W(b,a)=0$. The standard Multics notation for tree names is used.

a. DIR(X > Y)→BR(X > Y)→DIR(X)

b. Any DIR and any BR→any ASTE

c. ASTE(X > Y)→AST(X)

d. Any ASTE→AST.HT, SST.FS and any PSTE

e. AST.HT→PC

f. PC→AST.RL

g. Any data→APT

h. Any value $W(i,j)$, with $i \neq j$, which is not defined by (a) to (g) is: $W(i,j)=0$.

This hierarchy is also represented in Figure 4.

5. System Restrictions

We can see in Figures 2 and 3 that we have interferences between

DC and SC

SC and SC

SC and PC

The simplest system conventions that we could make to remove these interferences are as follows:

No segfault allowed in DC (solves DC - SC)

No segfault allowed in SC (solves SC - SC)

No page fault allowed in SC (solves SC - PC)

But these conventions are too restrictive. In fact, in SC we would like to use the PC machinery as much as possible; we accept the possibility of not using it in very special cases, in order to remove the interference, but, in general, we want to use it. The same way, in DC we would like to use the SC machinery, except when there is again a possibility of interference. Therefore, this solution, although simple, is not satisfactory.

A further analysis of DC, SC and PC has shown that the following restrictions are sufficient. They will be justified in the appropriate BG. section.

a. Page faults must be inhibited in a process while AST.RL is locked on its behalf.

b. Segment faults must be inhibited in a process while AST.HT, SST.FS, any PSTE or any ASTE is locked on its behalf.

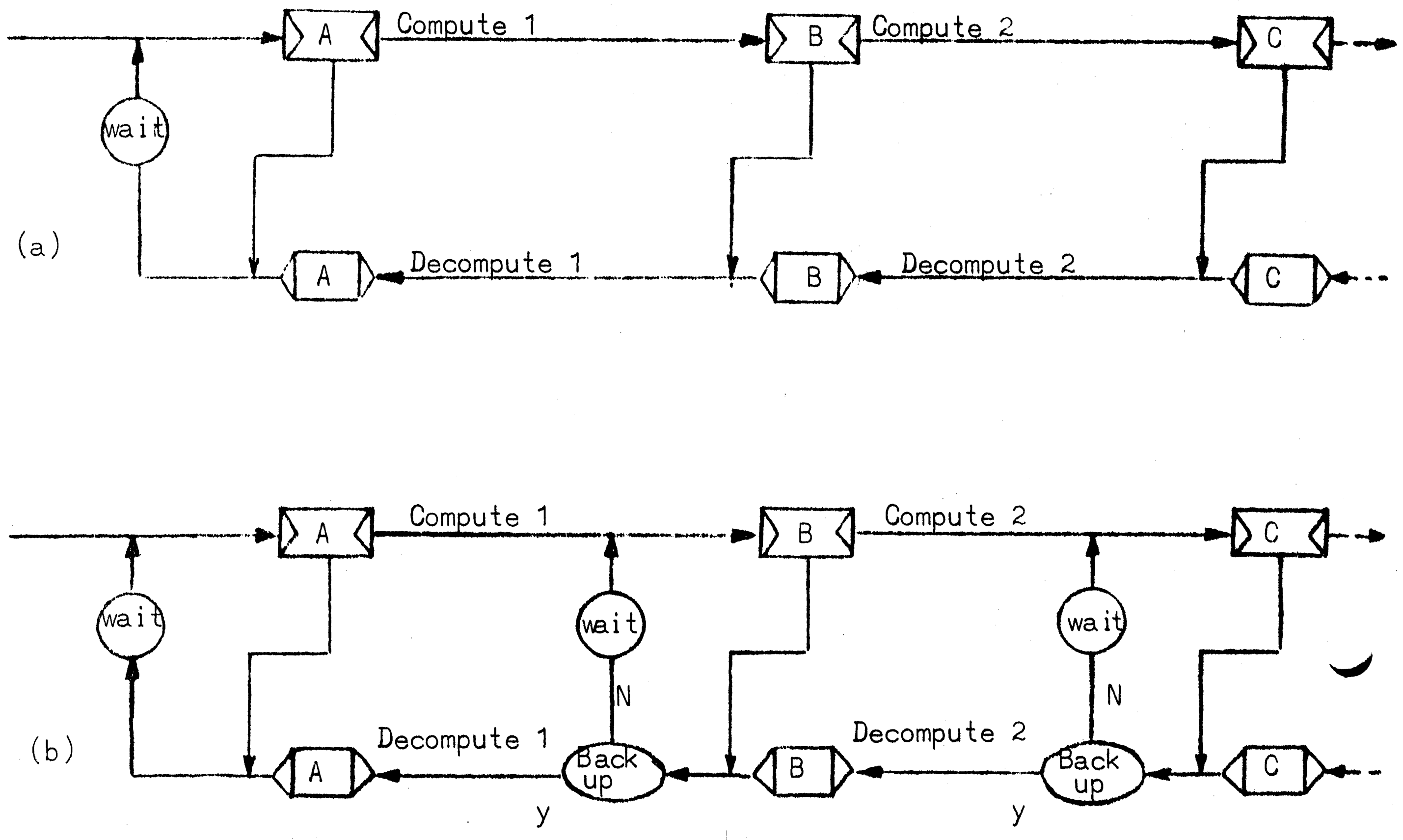
c. Segment faults are allowed in a process while some of the DC data are locked on its behalf, only if this segment (whose tree name will be designated X > Y) satisfies one of the following conditions:

- It is always active and its branch cannot be modified,
- BR(X > Y) and DIR(X) are already locked by this process,
- All DC data already locked by this process has the wait permit for BR(X > Y).

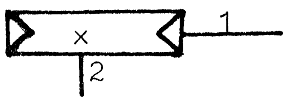
(a) is a minor restriction in SC. (b) is an important restriction in SC; it is practically equivalent to "no segment fault in SC". (c) is a minor restriction in DC since most of the directories referenced in DC satisfy one of the required conditions.

The following MSPM sections will deal with the enforcement of these conventions in page control, segment control and directory control.

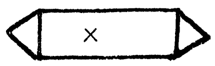
FIGURE 1



The notation used in Figure 1 is as follows:



= try to lock datum x; if successful, then take path 1 else take path 2

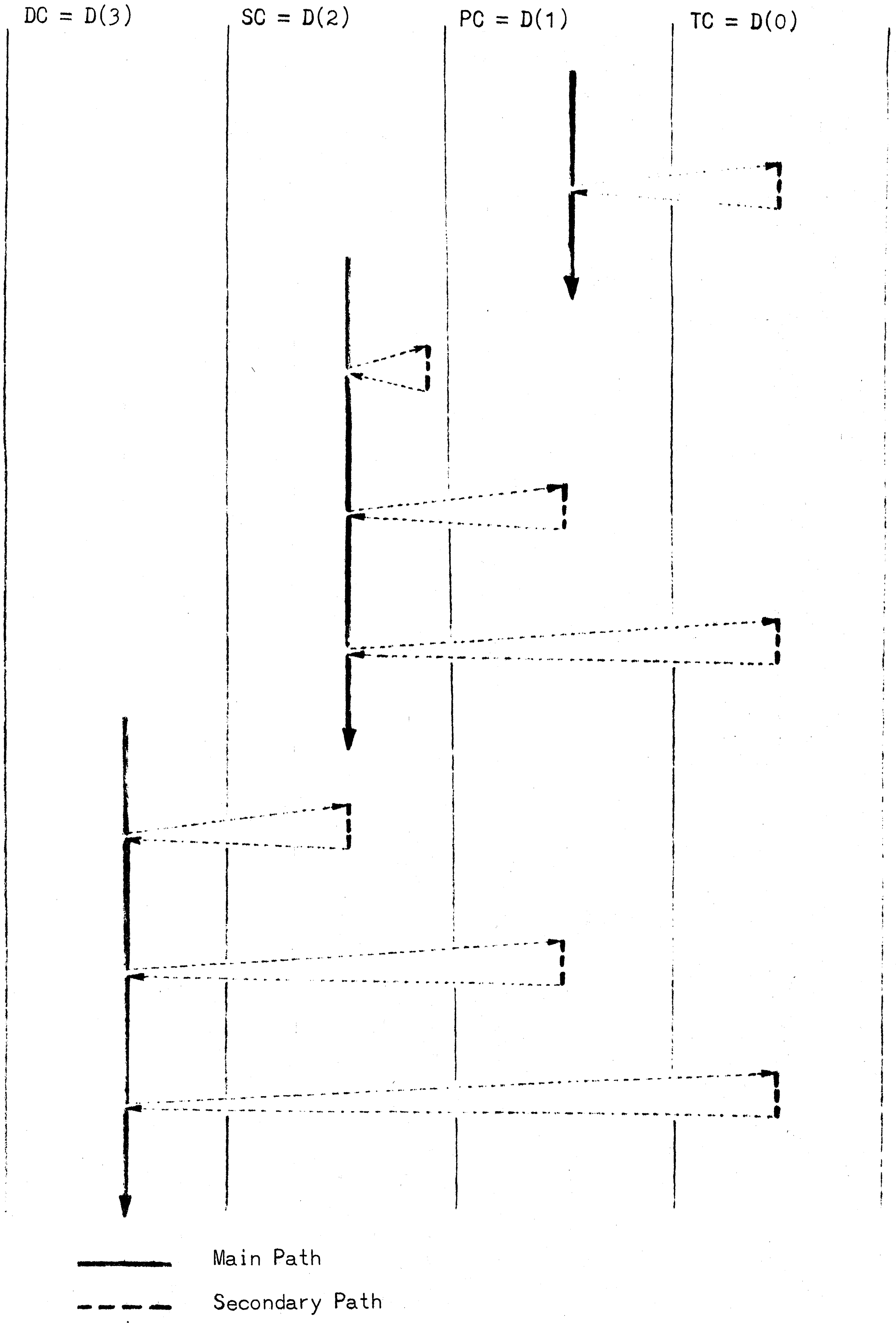


= Unlock datum x

compute	= Any set of instructions
decompute	= The set of instructions needed to initialize the process to the state it was before the "compute" operation
back-up	= "Back-up" is a boolean function whose value is "NO" if all data locked on the behalf of the process have wait permit for the datum we nned to wait for; otherwise its value is YES

FIGURE 2

Main and Secondary Paths



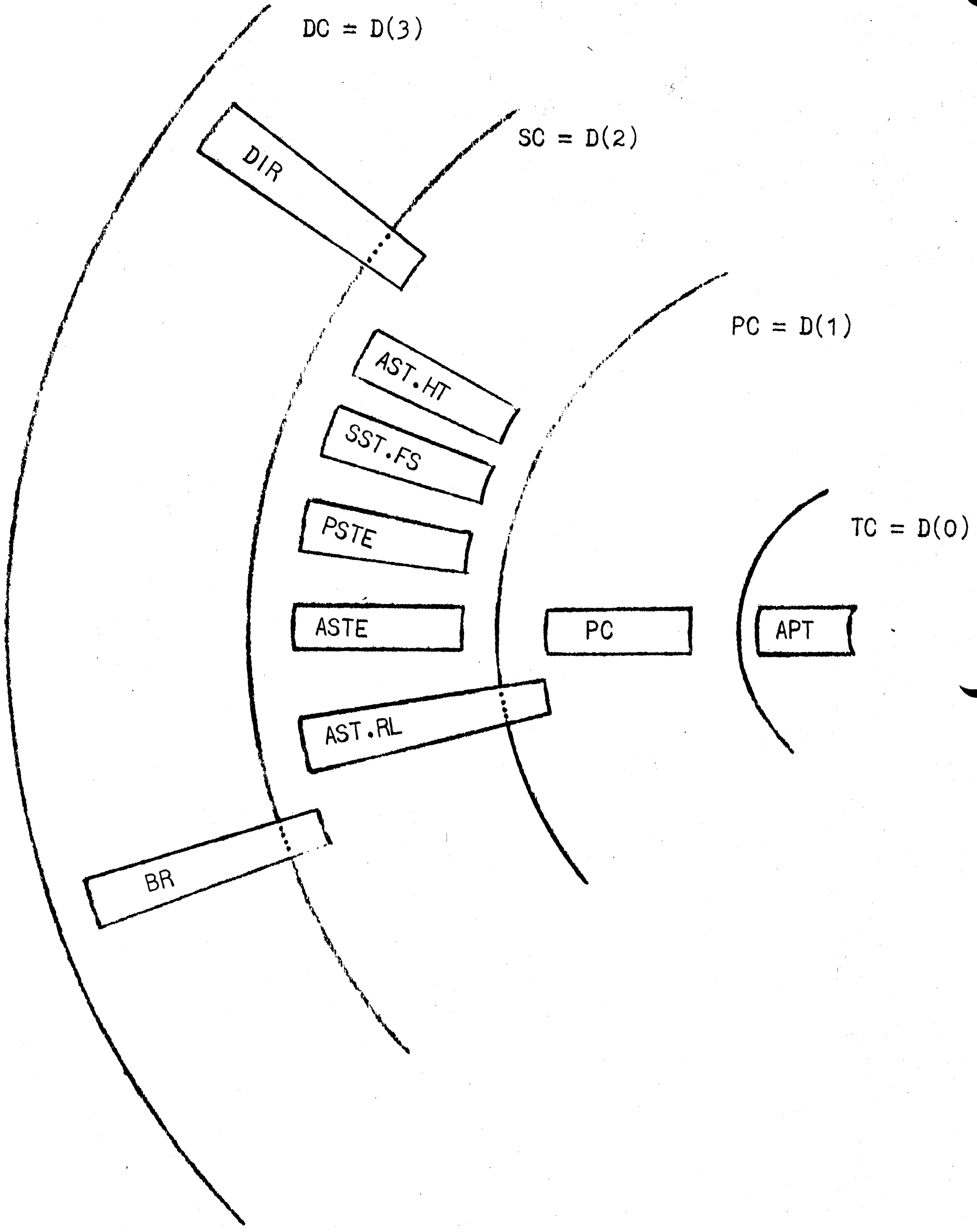
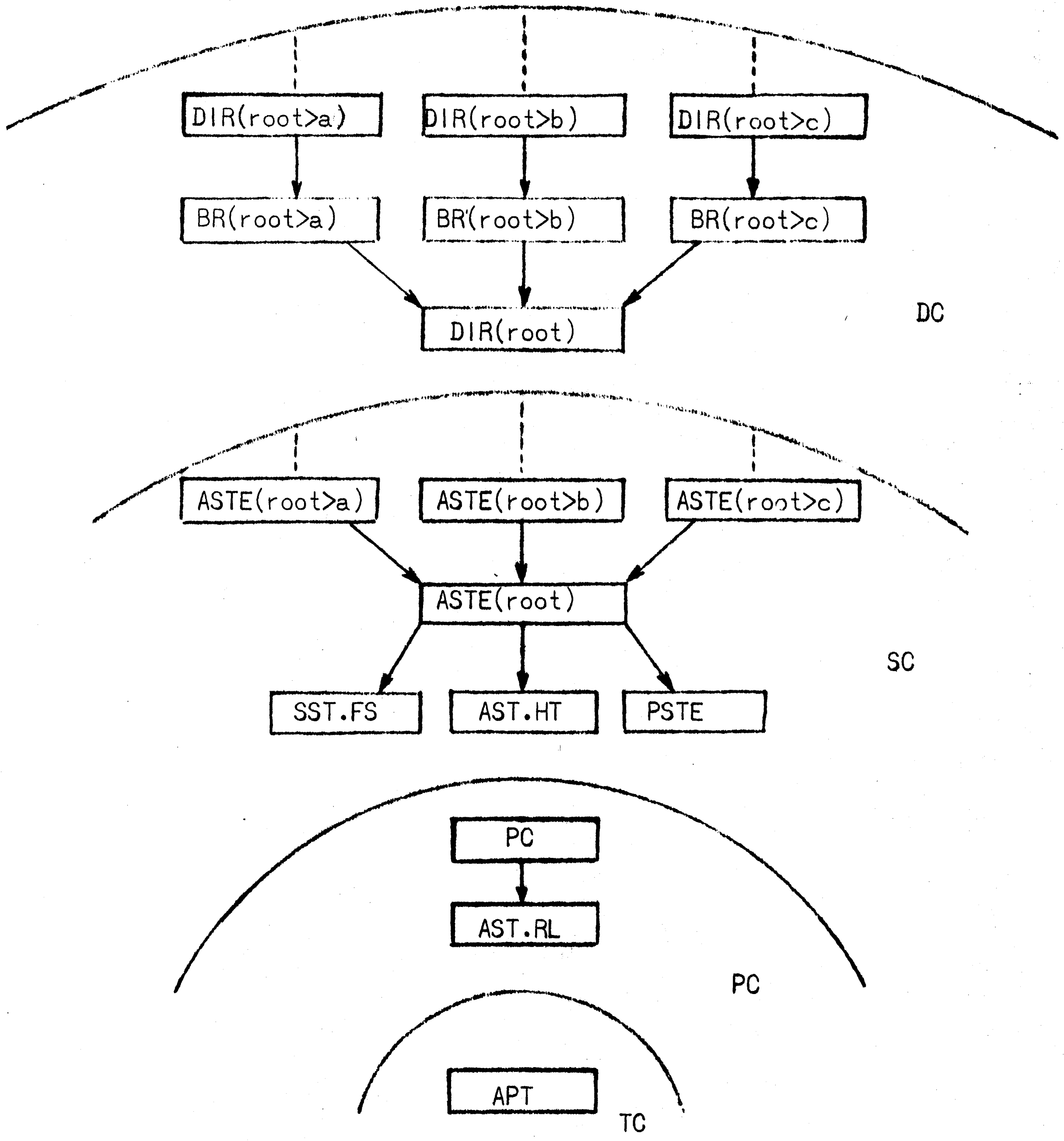


FIGURE 3

Data Needed In Each Domain

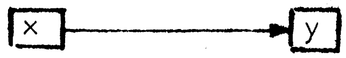
FIGURE 4

Multics Wait Hierarchy

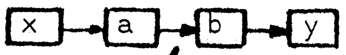


Datum x has wait permit for y if and only if one of these conditions is true:

1. There is an arrow pointing directly from x to y:



2. There is an arrow pointing indirectly from x to y:



3. x is in an outer ring: