Published: 3/01/67

Identification

User Control of Program-Device Synchronization.
J. F. Ossanna.

Purpose

The section describes the method of controlling the degree of synchronization between issuance of input/output requests to the I/O System (IOS) and the actual performance of physical input/output by the IOS. Internal synchronization management among the modules of the IOS is described in Section BF.2.25. Outer calls concerned with program-device synchronization and defined in this document are:

readsync
writesync
resetread
resetwrite
worksync
readwait
writewait
iowait
abort

In addition, a special use of the attach call for "emergency" attachment is described. An additional mode that can be used during conventional device attachment to suppress certain asynchronies is also described.

Program-Device Synchronization: General

Following the receipt of a request for input/output from a user program, an I/O System begins executing the sequence of steps necessary to ultimately accomplish the actual input/output. These steps may concern code conversion, intermediate buffering, physical record formatting, etc. Typically, control is returned to the user program before input/output is physically completed; final completion is accomplished or determined to have occurred at some subsequent time when the I/O System is again in control. A familiar example of delayed physical output occurs when output data is buffered until sufficient data has been collected to fill a physical record. It is also true that some of the steps necessary for satisfying input/output requests can be taken before the corresponding request from the user program. A familiar example of this case occurs when a physical record is read ahead of the time that all of its data is needed.

The relationships in time between user-program requests for input/output and the I/O System's efforts to satisfy these requests are complex. These relationships are usually determined by I/O System design. In contrast, the Multics IOS is designed to clarify these relationships and to provide the user with some control over them.

Loosely speaking, if the IOS's work to satisfy a request is not begun until the request is issued and the physical input/output is completed before control is returned to the user, the user program and the device are said to be operating synchronously. Otherwise, they are operating asynchronously. Rigorous definitions are given in later sections of this document.

The Multics IOS offers user control over three kinds of synchronization relationships - read, write, and workspace synchronization. Read synchronization control is concerned with whether or not read-ahead is performed by the IOS, and, if it is, how much read-ahead is performed. Write synchronization control is concerned with whether or not the IOS performs write-behind buffering or not, and, if so, how much. Workspace synchronization control is concerned with whether or not control is returned to the user program before the IOS is finished copying data from or storing data into the user's workspace. It should be observed that read-ahead and write-behind may each fall into two classes - avoidable and unavoidable. Some of the latter usually occurs during input/output on physical-record-oriented devices. User control is applicable only to avoidable read-ahead and write-behind. A detailed description of these forms of synchronization control including their interrelation is given in later sections of this document.

Usefulness of Synchronization Control

Historically, one of the major reasons for providing read-ahead and write-behind was the increased overall system efficiency resulting from the overlapping of program execution and input/output transmission. However, system efficiency could suffer when a user's access behavior differed from that assumed during buffer strategy design; for example, a user accessing a tape almost randomly needs only the minimum input/output buffering. For Multics, the efficiency argument for asynchronous input/output applies, but with less force. Instead, the reduction of the total elapsed time required for the completion of a program with significant input/output becomes the major reason for providing and controlling asynchronous input/output on noninteractive devices.

Control of input/output asynchronism is useful on interactive devices for additional reasons. For example, read-ahead on a typewriter is desirable generally and especially when large amounts of typing are involved. On the other hand, synchronous input/output is most important when the user is interacting with an undebugged, strange, or many-branched program.

Most of the discussion thus far has been concerned with read and write synchronization. The usefulness of the workspace synchronization control mentioned earlier stems from additional considerations. First, a user program may need to issue input/output requests at a rate greater than that possible with any ordinarily conceivable read-ahead or write-behind strategy. If the user program does not insist that its workspace be emptied (or filled) before return of control, it can receive control back sooner. An example of this need occurs when copying (with one pass) a record-gap-less tape using a multiple workspace strategy. Second, a user program may want to poll a series of devices without regard for which ones have data available. For example, a user program may be willing to accept data from either of two typewriter keyboards. If the user program does not insist on its workspace being filled before return of control, it can issue a read call for the second keyboard before any data is received from the first keyboard.

Device Dependence of Synchronization Control

The usefulness, variety, and possible degree of synchronization control is, of course, somewhat device- and situation-dependent. The manner and extent to which the IOS can or does comply with user synchronization control requests is necessarily dependent on the specific implementation of specific Device Interface Modules (DIMs) within the IOS. Because flexible and dynamic program-device association is a basic tenet of the Multics IOS, the response to and behavior following synchronization control requests is as uniform and equivalent as possible among the various DIMs. These requests are interpreted or tolerated by every standard DIM, and can almost always be safely issued. A possible exception occurs when workspace synchronization control requests are issued to achieve real-time advantages. The reader is referred to the various sections of MSPM Section BF describing input/output on particular devices.

Degrees of Input/Output Request Completion

To simplify subsequent discussion, certain terms and concepts relating to possible degrees of input/output request completion are discussed here. To begin with, whenever a user program receives control back following a read/write call, the request can be considered to be at least logically initiated. That is, the requested input/output will eventually be performed, if at all possible, without further intervention by the user program on behalf of that specific request. Of course, hardware errors, system failure, facilities reservation expirations, and other generally unexpected events can conspire to frustrate the completion of physical input/output. Also, a detach call is required in certain instances to force completion. The least amount of work that the IOS can do to achieve logical initiation is to place the call in a software queue.

The next degree of completion is logical completion. When the user program receives control back following a logically completed read/write call, two things are true: (1) the data has been transmitted to or from his workspace; and (2) certain status bits in the status bit string will reflect their final value (see

Section BF.1.21). A logically completed write call may not be physically completed; the output data may merely have been copied into an IOS buffer.

Another degree of request completion is physical initiation. When this stage is reached, the input/output has actually been queued in the GIOC hardware, and will proceed unless deliberately aborted by the IOS or by system or hardware failure.

The final stage of completion is physical completion. At this stage input/output has not only been accomplished by the hardware, but input data has been delivered to the user's workspace.

Logical and physical completion are each indicated by bits in the status bit string (see Section BF.1.21). In summary the four stages are:

(A) Logical initiation. (Software queued)
(B) Logical completion. (Data transmitted to or from the workspace)
(C) Physical initiation. (Hardware queued)
(D) Physical completion.

Return of control from a read/write call implies (A). (B) implies (A), and in the case of a read call, also implies (C) and (D). And (C) implies (A) but not necessarily (B).

Introduction to Workspace Synchronization

The subject of workspace synchronization control will be fully discussed in detail in a later section of this document. The present introduction is provided to simplify the following discussions of read and write synchronization control.

When a user program issues a read/write call to the IOS, one of the arguments specifies the user's workspace. This is the location within the user program where either input data is placed by the IOS or output data is removed by the IOS. The workspace synchronization mode is either synchronous or asynchronous. This mode is associated with a particular ioname. Normally the mode propagates down the i/o-path leading from the ioname (see Sections BF.1.03 and BF.2.25).

The workspace synchronization mode normally provided by the IOS is synchronous. The user must overtly ask for the mode to be made asynchronous by a call to be described in a later section of this document. In the normally synchronous workspace case, the user program receives control back after a read call with its workspace filled with input data or a write call with the output data copied from its workspace. In other words the IOS is finished with the user's workspace. A more rigorous statement is that control is returned after logical and/or physical completion, depending on the read and write synchronization modes.

If the write synchronization mode is asynchronous, the IOS returns control to the user program as soon as possible after it arranges for the read/write request to be done. Specifically, the return occurs after logical and/or physical initiation, depending on the read and write synchronization modes. Before the user can subsequently use his input data or reuse his output workspace, he must overtly determine the condition of the workspace by interrogating the completion bits in the status bit string. Alternatively, the user program can issue one of the "wait" calls described in a later section of this document.

Read Synchronization Control

The read synchronization mode is either synchronous or asynchronous. This mode is associated directly with the "device" and not with any intermediate streamnames used to reach the device. Specifically, this mode is a property of the device or frame terminal defined in Section BF.1.01, and more specifically it is a property of the primary device terminal defined in Section BF.1.03.

The call to set the read synchronization mode for a device is:

```
call readsync( ioname , rsmode [, limit [, status ]])
```

The brackets, [ ], indicate optional arguments and sets of arguments in the usual way. The first argument, ioname, is any ioname on an i/o-path leading to the desired device. rsmode is a one character string specififying the desired read synchronization mode. rsmode = "A" requests the asynchronous mode and rsmode = "S" requests the synchronous mode. limit is a pointer to a 35 bit signed integer which specifies the maximum permitted or recommended amount of read-ahead in terms of elements of the current element size. If limit is not provided or is a null pointer, no limit is specified and the IOS will impose a default limit, which is device- and possibly situation-dependent. status is a pointer to the status bit string (see Section BF.1.21).

If a read call is issued prior to any readsync call, the read synchronization mode is set to asynchronous.

Setting the read synchronization mode to synchronous prevents only "avoidable" read-ahead. Unavoidable read-ahead can occur. The most common examples of unavoidable read-ahead occur on physical-record-oriented devices such as magnetic tape and unit record devices. For example, a physical tape record may be read by the IOS to obtain data needed to satisfy a read request; if data in only part of the record was needed, the remainder was unavoidably read-ahead.

In the synchronous read mode, the IOS will not read data from a device until a read call is made. Then, the IOS attempts to read only enough data to satisfy the call. For example, if the device is a typewriter keyboard, the keyboard is not unlocked to permit typing until a read call arrives. Then the IOS typically reads until an appropriate break character arrives and locks the keyboard again. If the read call does not consume all the data read, some unavoidable read-ahead has occurred. The latter happens, for example, when the read call requests fewer characters than the number that actually preceded an erase/kill delimiter.

If a <u>readsync</u> call sets the read synchronization mode to asynchronous, the IOS begins to apply a device dependent read-ahead strategy. In any case the avoidable read-ahead is held to less than the current limit; the IOS actually attempts to hold the total read-ahead to less than the limit. If the device is a spontaneous provider of data, the IOS attempts to read the device continuously. A typewriter is an example of a controllable spontaneous device; the IOS attempts to leave the keyboard unlocked and collect data continuously except when output is being printed.

Unused data collected during read-ahead on a sequential, forward-only device is not discarded following a <u>readsync</u> call which sets the read synchronization mode to synchronous or which lowers the read-ahead limit below the amount read ahead. Such unused data is discarded only upon receipt of a <u>resetread</u> call.

If the workspace synchronization mode is synchronous, return of control following a read call necessarily implies physical completion. If the workspace mode is asynchronous, return of control following a read call implies (at least) logical initiation when the read mode is asynchronous, and physical initiation when the read mode is synchronous. These relationships are depicted in the following table (where A = Asynchronous, S = Synchronous, P = Physical, L = Logical, I = Initiation, and C = Completion):

		<u>Workspace Sync Mode</u>	
		A	S
Read)	A	LI	PC
Sync)			
Mode)	S	PI	PC

A call is provided for discarding existing read-ahead data; it is:

```
call resetread( ioname [, status ])
```

ioname and status are as defined for the readsync call. Issuing the resetread call does not affect the read synchronization mode.

Write Synchronization Control

The write synchronization mode is either synchronous or asynchronous. This mode, like the read synchronization mode, is associated with the device. The call to set the write synchronization mode is:

```
call writesync( ioname , wsmode [, limit [, status ]])
```

The first argument, ioname, is any ioname on an i/o-path leading to the device. wsmode is a one-character string specifying the desired write synchronization mode. wsmode = "A" requests the asynchronous mode, and wsmode = "S" requests the synchronous mode. limit is a pointer to a 35 bit signed integer which specifies the maximum permitted or recommended amount of write-behind in terms of elements of the current element size. If limit is not provided or is a null pointer, no limit is specified and the IOS will impose a default limit, which is device- and possibly situation-dependent. status is a pointer to the status bit string (see Section BF.1.21).

If a write call is issued to the device prior to any writesync call, the write synchronization mode is set to asynchronous.

Setting the write synchronization mode to synchronous prevents only "avoidable" write-behind. Unavoidable write-behind commonly occurs on physical-record-oriented devices. For example, the writing of a physical tape record is ordinarily delayed until sufficient output data has been collected to write a complete record of the standard size.

If the write synchronization mode is asynchronous, the IOS employs a device-dependent write-behind strategy. The avoidable write-behind is held to less than the limit; the IOS actually attempts to hold the total write-behind to less than the limit. Physically-initiated output is regarded as a part of the write-behind data, until completion occurs. Before returning control to the user following a write call, the IOS will have logically initiated the write request and, if the workspace synchronization mode is synchronous, copied the output data from the user's workspace. If output was not already proceeding because of previous write calls, the IOS may physically initiate part or all of the output. If a write call is received having a workspace size which would cause the current quantity of write-behind to be incremented beyond the current limit, the IOS responds in either of two ways, depending on whether or not the amount by which the limit would be exceeded is less than or greater than a device-dependent allowable extension limit. If the excess is less than the allowable extension, the only effect is that the return of control to the user is delayed until the write-behind again falls below the current limit. If the excess exceeds the allowable extension, the write call is rejected. If a writesync call lowers the limit to an amount below the current write-behind, and a write call occurs before the write-behind drops below the new limit, this write call is treated exactly as though its request had caused the excess.

If the write synchronization mode is synchronous, the IOS returns control to the user following a write call after the requested output is physically completed (except for unavoidable write-behind) when the workspace synchronization mode is synchronous or after the requested output is physically initiated when the workspace mode is asynchronous.

The relationships between the write and workspace synchronization modes are depicted in the following table (where the letters A, S, P, L, I, and C were defined in connection with an earlier similar table):

		Workspace Sync Mode	
		A	S
Write) Sync)	A	LI	LC
Mode)	S	PI	PC

A call is provided for discarding existing write-behind data; it is:

call resetwrite(ioname [, status])

ioname and status are as defined for the writesync call. Issuing a resetwrite call does not affect the write synchronization mode. Physically-initiated output is not discarded. The status bit strings of the affected write calls will reflect the occurrence of the resetwrite call; a bit is set indicating that the call was aborted by user request prior to physical completion (see Section BF.1.21). A partially-affected write call will have the same bit set even though part of the data may have been written out.

Workspace Synchronization Control

The workspace synchronization mode is normally synchronous. The call to control this mode is:

call worksync(ioname , wkmode [, status])

The first argument, ioname, is the ioname at which the mode is to be established. status is a pointer to the status bit string. wkmode is a one-character string specifying the desired workspace synchronization mode. wkmode = "A" requests the asynchronous mode, and wkmode = "S" requests the synchronous mode. Typically, the outer module at ioname reissues the worksync call to the next outer module along the i/o-path; thus this mode usually propagates down the entire i/o-path. If a second i/o-path merges with an i/o-path at an ioname at which the workspace synchronization mode is asynchronous, modules in the second i/o-path may receive return of control back prematurely after a read/write call. This occurrence is detectable by examining the status bit string. It can also be prevented by interpolating a Monocaster (see Section BF.2.18), which detects premature returns and issues iowait calls to reestablish workspace synchronism. If two merging i/o-paths are both workspace asynchronous and one is set to synchronous, the common portion of the other is affected; the resulting effective workspace mode of this other path has in most instances also become synchronous.

When the workspace synchronization mode is asynchronous, the user program can get control back following a read/write call before logical or physical completion of the request. The user program can subsequently determine the request status by interrogating the completion bits in the status bit string. Alternatively, the user program can issue one of the following "wait" calls:

```
call readwait( ioname [, status ])

call writewait( ioname [, status ])

call iowait( ioname [, transptr [, status ]])
```

ioname and status are as defined for the worksync call. transptr is a "transaction" pointer used to identify a previous transaction (see Section BF.2.02). transptr is literally the status pointer of the earlier call which is of interest. For transptr to be a meaningful transaction pointer, the "hold" bit in the corresponding status bit string must have been set following the return from the call of interest (see Sections BF.1.21 and BF.2.02). A nonmeaningful transaction pointer causes an error indication in the status bit string of the "wait" call. The readwait and writewait calls implicitly refer to the most recent read and write transactions respectively. Return of control to the user program following a readwait, writewait, or iowait call occurs when the referenced transaction has been completed to the degree required to enable a return had the workspace synchronization mode been synchronous. If transptr is absent or is a null pointer, the transaction referenced in the iowait call is the most recent read or write call. These "wait" calls are inappropriate when the workspace synchronization mode is synchronous, and their use will cause an error indication in the status bit string of the "wait" call.

Method for Aborting Transactions

A mechanism is provided for cancelling or aborting previously-issued read and write requests and can be used irrespective of the workspace synchronization mode. Only read and write calls are affected and any interpolated non-read/write calls have their normal effect. The call for aborting transactions is:

```
call abort( ioname [, transptr [, status ]])
```

ioname is any appropriate ioname, and status is a pointer to the status bit string. transptr is a transaction pointer as described earlier in connection with the iowait call. The abort call requests cancellation of the read/write call to which transptr refers and of all subsequent read/write calls preceding the abort call. If transptr is missing or is a null pointer, the IOS attempts to abort all existing read/write calls which are physically incomplete. If transptr refers to a physically-completed request, the IOS still attempts to abort the subsequent requests. Successfully-aborted read/write calls have a status bit set indicating that the call was user-aborted. Unsuccessfully-aborted calls are merely indicated to be physically complete. Partially-aborted write calls are considered to be aborted. The status of the abort call contains an "unsuccessful" indication, if and only if the IOS was unable to cancel any outstanding requests.

Read/Write Asynchrony Prevention

Under certain circumstances it may be useful to operate a particular device with a synchronous read and/or write mode while using a program written to specify or permit asynchronous read and/or write modes. This may be achieved at device-attachment time by including a special mode character in the mode argument of the attach call or at any time by use of the mode in a changemode call (see Section BF.1.01).

The special asynchrony prevention character is "Y". The presence of the sequence "RY" in mode sets the read sync mode to synchronous and causes future readsync calls to be ignored; similarly "WY" sets the write synchronization mode to synchronous and causes future writesync calls to be ignored; "Y" unpreceded by "R" or "W" forces both modes to be synchronous. The subsequent use of "notY" in these same contexts causes future readsync and/or writesync calls to have their normal effect, although the modes are not changed at the time of the changemode call. The "not" is used here as a representation of the PL/I "not" symbol (ASCII "overline").

No mechanism is provided for advance prevention of workspace asynchrony, inasmuch as programs use the asynchronous workspace mode for purposes (outlined earlier) likely to be more or less incompatible with synchronous workspace operation.

Emergency Device Attachment

Under certain conditions it may be desirable or necessary to suspend in a restartable way previously-requested input/output which is currently in progress on a particular device. For example, the Supervisor may need to write an emergency message on a typewriter, advising of an automatic logout or other imminent happening. A special use of the attach call is provided for this purpose. An "emergency" device attachment may be made by including the mode character "E" in the mode argument of the attach call. Such an attachment overrides the existing device attachment and causes suspension of any current input/output on the device. A new independent series of input/output transactions may now be undertaken. Subsequently, a detach call with disposal containing "E" will end the emergency attachment and reinstate the previously-suspended input/output.

An emergency attachment-detachment sequence effectively permits the instant interpolation of an independent series of input/output transactions during the course of existing input/output. Such emergency sequences may be nested to any depth. The IOS accomplishes this independent input/output by using a new and separate i/o-path and attachment graph below the device terminal within the IOS (see Section BF.1.03). The separate, independent attachment and transaction graph is normally obtained by creating a new Device Manager Process. The previous Device Manager halts upon receiving a please-quit request from the IOS in the process in which the emergency attachment was requested.

Possibly the most common example of emergency attachment that will occur results from the receipt of a valid "quit" signal from a device which is also the current command source (see Section BF.3.01). An emergency attachment is made to this source to accomplish the command input/output.