Superseded by new IPC

Published: 01/19/68
(Supersedes: BG.15.01, 06/03/66)

not yet implemented 10/22/68

Identification

Process Wait and Notify Module
M. R. Thompson, P. Schicker

Purpose

The Multics Traffic Controller (see Section BJ.) furnishes the basic tools for allowing processes to cease execution temporarily (block) and resume execution subsequently (wakeup). The Basic File System offers a group of primitives for the purpose of coordinating the employment of these Traffic Controller functions when they are used by File System and other ring-0 procedures. These primitives comprise the Process Wait and Notify Module (PWN). The PWN keeps a list of the processes that are blocked or about to be blocked as a result of encountering locks or other forms of impasse in the Basic File System. When the PWN is notified that some event has occured, it sends a "wakeup signal" (see BJ.7.00) to all processes that are waiting for that event.

Overview

The need for coordination of File System use of the Traffic Controller is a consequence of the distributed supervisor approach taken in Multics. That is, shared File System procedures can be operating in behalf of a given process, say A, when it becomes necessary to suspend execution of A owing to what amounts to a "private" File System issue (e.g., waiting for the arrival of a page, or for the unlocking of a directory), as opposed to the sort of blocking which is somehow of A's "own volition" (e.g., A is part of a user-process-group in which parallel processing is being undertaken and must wait for the completion of a particular chore by another process, say B). The File System is of course responsible for assuring the subsequent resumption of A's execution in such cases. However, the File System can also encounter a similar issue requiring blocking when operating in behalf of another process, say C. Therefore, a mechanism must be provided which will enable A's wakeup and C's wakeup to be handled correctly when they occur. This mechanism is provided the PWN and its (system-wide) data base, the Process Waiting Table (PWT). So instead of calling block directly and assuming direct responsibility for arranging a wakeup, File System

routines call the addevent primitives to have the events they are about to wait for entered on appropriate lists of events in the PWT, and then call the wait primitive, which will call block (unless the event has already happened - see below). Then those procedures which recognize the events (e.g., the iodone routine in Page Control) simply call the notify primitive, which will direct wakeups to any processes which are waiting for the event in question.

To avoid the situation in which the call to block is made for a process after the event in question has taken place, the following strategy is employed: Each PWT entry contains a "done switch". The notify primitive sets this switch to "on" when it processes the event and state variable occurence corresponding to the entry, so that wait can check it when called and will only proceed when the done switch is "off". Further details are given in the individual description of the primitives, below.

Process Waiting Table (PWT)

The process waiting table (PWT) is the data base for the PWN and consists of lists of all the processes that are currently waiting and the events for which they are waiting. Since it is convenient that the number of events be fixed, certain events may require a state variable to further define why the process is waiting. For example, an event might be the availability of a directory which was previously locked, and its state variable would identify which directory; or the event might be the arrival of a page in core, and its state variable would identify what page. A complete list of events and their state variables may be found below.

The PWT consists of threaded event lists each of which contains a list of all the processes that are waiting for a particular event. The PWT is divided between two segments, one which is wired down in core and one which is a normal (paged) segment. The event lists that might be used during the handling of a page fault must be kept in the wired down segment. Examples of such events are waiting for a page to come into core or waiting for a data base that is used on page faults to become available. Those event lists which do not have to be kept in core are kept in a normal segment. The event numbers are chosen

so that all numbers less than a given constant belong to the wired down part of the table, and all numbers greater than this constant belong to the normal part of the table. The two parts of the table will be structured and manipulated in the same manner with the exception of the action to be taken when an event list lock is encountered as explained below. Each part of the table also contains a threaded list of vacant entries, and the value of the highest location used in that part of the table. Each entry in an event list consists of the process id, the state variable, and a pointer to the next entry. The pointer to the top of each list is found at the beginning of the table.

In the event that either part of the PWT becomes completely full, the process that is trying to enter the table will be forced to loop through a delay routine. The routine insures that the processor is unmasked since the process is no longer trying to lock a data base (a calling routine or the PWN itself may have masked the processor before attempting to lock the data base). It then calls each DIM in order to speed their task of getting pages in and out of core (this action causes processes to be removed from the PWT). The delay routine then restores the original value of the processor mask and returns. After each return from delay the process again attempts to place itself on the PWT. The PWT is sufficiently large so that it will rarely become full.

Since the PWT is a common data base, it may itself have to be locked. In order to avoid conflict when changing pointers, each event list must be locked when it is being searched, when a process is being removed from it, or when a process is being added to it. When a process encounters a locked event list on the normal PWT, it can proceed as it would for any ordinary data base; that is, enter itself on the appropriate event list on the wired down PWT, and go blocked. The appropriate list on the wired down PWT is the list of processes waiting for an event list on the normal PWT to become available, and the state variable is the number of the locked list. However, it is possible for this wired down list to be locked also, but now there is no other list on which the process can be entered. Thus, when a process encounters a locked list on the wired down PWT, it does not attempt to go blocked, but instead loops on the lock-testing instruction until the list is unlocked. Since any process that has a wired down PWT list locked for it also has the processor masked, the time that the process keeps the list locked is short. There is a special locking routine (loop-lock) for the PWN to use when it attempts to lock an event list on the wired down PWT.

In general, the list_handling strategy of this module is as follows. New processes are added to the top of event lists, and event lists are searched from top to bottom to find processes. The entire event list is searched, and all the processes to be signaled because of one event are found before the traffic controller is called to actually wake them up. The top of the vacant entry list can be locked so that two processes do not conflict when they use or replace vacant entries. Vacant entries are used and replaced from the top of the vacant entry list. If the vacant entry list is empty, entries are used from the higher unused part of the segment. In order to keep the list consolidated, when an entry is vacated its location is compared to the highest location currently used and if it is equal the highest used location is decreased by one. If its location is lower, it is added to the vacant entry list.

Contents of the PWT

1. Top Pointers

   A. Event 1 list information

      1. Pointer to top of list
      2. Interlock for list
      3. Count

   B. Event 2 list information

      1. Pointer
      2. Interlock
      3. Count

   .
   .
   .

   X Event n list information

      1. Pointer
      2. Interlock
      3. Count

   Y Vacant entry list information

      1. Pointer to top of list
      2. Interlock for top of list
      3. Highest relative location used in the table

II. Entry for event list

A. Pointer to next entry
B. Pointer to previous entry (=0 if head of list)
C. Process indentification
D. State variable
E. Done switch
F. List number

III. Entry for vacant entry list

A. Pointer to next vacant entry

Primitives

There are four primitives provided by the PWN. They may be used by any file system module. Each of these primitives masks the processor before locking any list on the wired down PWT, and restores the previous mask as soon as the list is unlocked. The primitives are:

```
pwn$addevent   (n,var,ind)
pwn$delevent   (n,ind)
pwn$notify     (n,var)
pwn$wait       (ind)
```

1. The primitive addevent enters the current process id and state variable in the waiting list for event n and returns.

```
call pwn$addevent   (n, var, ind);
```

n: event number for which process is waiting
var: state variable for this event
ind: index into PWT (if ind<0 then |ind|=index into normal PWT else |ind|=index into wired down PWT.). Returned by addevent.

The addevent routine first tests and when possible locks the vacant entry list, gets a vacant entry from the top of the list, rethreads the list top pointer and unlocks the vacant entry list. It can then place the process id and var in this entry, test, and when possible lock the event list, add the entry to the top of the specified event list, rethread the top pointer to this list, unlock the event list, and return to the caller.

2. The delevent routine removes the entry indexed by ind from list n and returns control to the calling procedure.

```
call  pwn$delevent (n,ind);
n:    event number
ind:  index into PWT
```

The delevent routine must test the lock on event list n, and when possible lock the list for itself. It then removes the entry indicated by ind from the list, rethreads the list, unlocks the list, adds the vacant entry to the vacant entry list, and returns.

3. The primitive notify checks the event list to find all the processes waiting for this event and state variable value. After setting a "done switch" in the corresponding PWT entry or entries, it calls the traffic controller to wakeup each of these processes.

```
call  pwn$notify (n,var);
n:    event number that has occurred
var:  state variable for this event
```

The notify routine must check the lock on the event list, and when possible lock it for itself. It then searches the list for a state variable value that matches var and when it finds one, notify remembers this entry, turns the done switch on, and continues to search the list.

When the entire list has been scanned, it is unlocked. The Traffic Controller is then called to signal all the processes which were found in the waiting list. When the Traffic Controller returns to notify, notify returns to its caller.

4. The primitive wait calls the Traffic Controller's entry block unless the specified event has happened.

```
call  pwn$wait (ind);
ind:  index returned by addevent
```

The wait routine checks the done switch in the entry specified by ind. If the switch is off, i.e. the event has not yet happened, the Traffic Controller's entry block is called. Upon return from block the switch is tested again and block is called again unless the done switch was turned on meanwhile. Whenever wait finds the done switch turned on, it deletes the entry and adds the now vacant entry to the vacant entry list, and then returns to its caller.

List of Events

	Number	Event	Used By*	Status Variable
Wired:	1	PWT locked	PWN	
	2	AST entry not available	PC,SC	AST pointer
	3	DST entry not available	PC	DST pointer
	4	IO count = zero	PC	AST pointer
	5	PTW not in service	PC	page number
	6	SST space unavailable, or	SC	"01"b for SST space available, or
		AST hash table locked	SC	"10"b for hash table unlocked
(Constant plus:)				
Unwired:	1	CACL locked	DC	directory unique ID
	2	directory locked	DC	directory unique ID
	3	entry locked	DC	entry unique ID
	4	active info locked	DC	entry unique ID

*PC = Page Control, SC = Segment Control, DC = Directory Control

PL/I Specification for PWT

```
dcl  1                pwt based (pwtptr) ,
          2           n_list fixed ,
          2           n_entry fixed ,
          2           max_entry fixed ,
          2           pwteptr ptr ,
          2           list (0:pwtptr->pwt.n_list) ,
                3     lock bit (36) ,
                3     ( head,tail ) fixed ,
                3     count fixed ,
       1              entry (pwtptr->pwt.n_entry) based (pwteptr) ,
          2           pid bit (36) ,
          2           var bit (36) ,
          2           ( next,last ) fixed ,
          2           dsw bit (1) ,
          2           evt fixed ;
```