Identification

Overview of Process Wait and Notify

R. L. Rappaport, Michael J. Spier

Purpose

Processes executing in behalf of the system in the hardcore ring sometimes have to abandon their current processor and revert to the "waiting" state, waiting for some system-event (such as the unlocking of a systemwide data base, or the arrival of a page from the drum) to happen. The Traffic Controller subroutines wait, notify, and addevent ~~and delevent~~ provide the mechanism by which a process can either enter the waiting state or release some other process from that state and put it back on the ready-list (notify).

Introduction

The Traffic Controller's interprocess communication entries block and wakeup (see BJ.3) provide the means necessary in order to allow a process to either give its processor away or to restore another process into the ready-list. Theoretically, subroutines block and wakeup would suffice to handle all cases of process wait and notify; however, experience has shown that block and wakeup, in themselves, are too primitive, functionally, and that to have a Process Wait and Notify (PWN) mechanism that makes use of these primitives was too costly from an efficiency point of view.

To be specific, the purpose of a PWN module is to "keep book" of all waiting processes, to remember which process is waiting for what event and to make sure that whenever a ~~specific~~ certain event happens, only the processes that are actually waiting for that event be "notified." PWN is concerned with "system events" (e.g. ~~waiting for system tables to unlock, waiting for a page to arrive from the drum~~ the unlocking of a system table, the arrival of a page from the drum) which are guaranteed to happen within a predictable period of time, normally measured in milliseconds. The main reason for having a process go into ~~a~~ the waiting state is that of maintaining efficient processor-resource management, the cost of going to wait must be fractional compared with the amount of processor time saved. An additional characteristic of system events is that a process will never wait for more than a single system-event at a time. Consequently, the cheapest way to provide a PWN facility is to thread the waiting-process' APT entry into a wait-list associated with a specific event, and to associate the head of that list with the event name so as to allow the notifying process to find the list and restore the waiting processes into the ready state.

The Active Event Table (AET, see BJ.1.3) is a table containing a group of relative pointers to a collection of event threads running through the APT. It is a wired-down table in segment <tc_data> and its size is an agreed-upon constant (actually a prime number, to facilitate hash-lookup.)

Let us suppose that there are N entries in the table. Events are communicated to PWN by name and all processes waiting for an event named A will be hanging off the thread pointed at by an AET entry associated with A. In order not to have to provide a unique AET entry for each possible system-event, we must perform a mapping from the set of all possible event names into the set of integers from 1 to N (table size.) In this way, processes waiting for events A and A' may both be placed on the same thread and may be notified incorrectly that their event has occurred. However, if the AET table size is large compared to the number of loaded processes (which alone may wait for a system-event) and the mapping from event names to numbers is done so as to evenly distribute the names over the numbers, the conflicts should be kept to a minimum.

Also, for reasons of efficiency, care is taken to insure that processes waitin for system-events will not be unloaded by the Traffic Controller Daemon Process (see BJ.6.)

The above-described strategy is designed to assure the smoothest and most efficient processor-resource management whenever a process is executing in the hardcore ring; this is of utmost importance considering the fact that a process may spend an estimated half of its virtual processor time in ring 0.

Implementation

An entry in the AET contains two items, a pointer to the head of the event list (which may assume a zero-value if there is no list) and a flag (which may assume one of the two values ON/OFF).

An AET entry is said to be inactive if its pointer is of zero-value and its flag is set to off; otherwise it is said to be active.

inactive = (pointer=0) & (flag=off)

active= ¬inactive

Depending upon the state of an entry, the four PWN subroutines operate according to the following algorithm:

ADDEVENT (A)	always	sets entry A's flag to ON
~~DELEVENT (A)~~	~~always~~	~~sets entry A's flag to OFF~~
WAIT (A)	if A inactive:	returns
	if A active:	puts itself on thread.A, abandons processor
NOTIFY (A)	if A inactive:	returns
	if A active	de-activates A, puts all waiting processes on ready list

A typical way of using the PWN facility is outlined below:

At some point in a computation we reach a point where we do not wish to continue until a particular condition is satisfied. Therefore we perform a test to see if the condition is satisfied; if yes, we simply continue and if not we arrange to wait for the condition to change in the following way. Starting from the original test:

1. Test condition. If true go to step 5.
2. If not true call addevent to activate event.

3. Test condition again. If still not true call wait and upon return go to step 1.

4. If the retest was successful call ~~delevent to reset the flag associated with this event.~~ notify to deactivate the event (and possibly wake up waiting processes). ~~This will deactivate the event if our process is the only one currently interested in it~~ and is necessary because we have a limited wired-down data base to keep active events in.

5. continue.

Sometimes in a computation we become aware of a condition of which others may be interested. In this case we call notify to "broadcast" the good news.

The test of whether or not an event is active provides an interaction between subroutines wait and notify. Without it, it would be possible for a process to put itself on an event-list and give its processor away <u>right after</u> that event was notified by some other process. If this was the last time for this event ever to occur, the waiting process will never again run. According to the PWN algorithm, notify deactivates the event. A process calls wait only after it has previously called addevent, which activates the event. When the process calls wait and finds the event inactive, it knows that someone has notified all the processes waiting for that event and consequently immediately returns.

The ~~four~~ three PWN subroutines are described in detail in sections BJ.2.1-3, respectively.

<u>Performance</u>

The approximate time necessary to execute a PWN subroutine is as follows (times are given in microseconds):

ADDEVENT	~~270~~	~200
~~DELEVENT~~	~~250~~	~~~200~~
WAIT (event inactive)		~200
(event active)		~700

NOTIFY (event inactive) ~200

(event active, one process waiting)~300