Kernel Introspection:Design Meeting 2007-04-20#

Who#

bstecher, cbugess, adanko, dbailey, mkisel, shiv

Summary#

What we did#

- we found and fixed a bug in the design for the deadlock smoke alarm
- new design for CPU hog detection
- more on reading bulk data from Proc
- more on the Generic Notifier

What else we need to do:

- create authoritative list of blocking states for which we will set timers for deadlock detection.
- Colin will advise what pathnames to use for the general notifier and bulk proc data transfer.
- we should all combine lists of things for which we should have notification thresholds, starting with the RLIMITS

Deadlock Detection#

Problem#

A particular customer is likely to have several threads that are forever receive blocked. We would be sending their HA controller a sigevent every minute for each of them. Bletch.

Proposal A13#

- Kernel adds default timers to only a carefully selected set of blocks: only those blocking states which can trigger a detectable deadlock.
  - only some deadlocks are detectable. For example, while two threads can deadlock on a pair of semaphores, any other thread could conceivably release either semaphore.
- Implementation will take a set of states, in a parameter, for which we will create default timers.
- but we must advise the customer to use a set of states that we can authoritatively say will detect all deadlocks that their HA controller can deal with.
- These timers only fire once. We assume that if the event, of a thread being blocked for 1 minute on a thread, is a true symptom of a deadlock, then the customer's HA controller will find and fix it. If it is not a symptom of a true deadlock, then that blocked thread cannot possibly trigger a later deadlock -- not at least until it unblocks and blocks again. On the second block, we will set another timer.
- The sigevent should identify the blocking thread id, so the customer's HA controller can poll for data focused on that thread

- We are likely to deliver sigevents in bursts to the customer's HA controller. We should advise the customer to collect sigevents received in some reasonable time (say 30secs or a minute) and then process them all at once.
- we still need an optimized call to read out all the sync information when the customer is triggered to poll.
    - perhaps we should have an API that takes a list of tids as a parameter, and returns the syncs for each of them. (i.e. bulk transfer) *'Note this is not quite compatible with the current bulk read proposal.*'

This design must meet these tests to be useful

1. in practice, we will not be sending the customer's HA controller a sigevent every minute or more. (Or this is no better than polling)
2. we will send a sigevent whevenver any deadlock (that the customer's deadlock-detection algorithm is capable of dealing with) occurs.

CPU Hog Detection#

Problem#

APS doesn't have a notification mechanism for either the system running out of CPU time, or a single partition using more time than some limit. (The only notification mechanism in APS is for running out of critical time which is specific to designated threads. So critical time notification doesn't help us solve this problem.)

Proposal A14#

- use APS to throttle back hogs so they do not harm the rest of the system
- advise the customer to do its custom hog detection and recovery at the 5 minute polling interval they would normally use for recording CPU usage history.
- we still require an efficient interface to allow all processes CPU usage to be read at the 5 minute polls.

This assumes that the customer's HA controller's first purpose in detecting CPU hogs is to limit damage to the rest of the system and that their second purpose is to kill and recreate them. (Kill/recreate in the hopes that the error state will be reset.) In which case we conjecture that customer's HA controllers can afford to wait 5 minutes to kill a hog if that hog is being calmed by APS.

This assumes that 8 (or possibly 16) partitions is enough.

Reading bulk data from Proc#

We believe there are two ineffeciences in reading all pid/tid data with individual calls:

1. traversing mapinfo structures (in the current implementation) just to get total memory used for one pid.
2. traversing the filesystem for each pid/tid

Proposal A15#

First, change allocators to keep a running total of memory used for each process so that no mapinfos need be traversed during calls to Proc for retreiving process info. We would return total memory used by a process as one of the RLIMIT values when returning data about rlimits or returning complete data about the process.

Second, create an optomized call for reading data about many pids/tids at once:

1. user needs only one open() to read data for all pids/tids.
2. allow many reads per open and return data about many pids and tids per read()
3. allow user to filter data returned, focused on task. (example deadlock detection means read all syncs for all threads; cpu hog detection means reading process cpu times for all processes.)
    - filtering is useful only if we expect the customer's HA controller to not simultaneously be doing deadlock detection and cpu hog detection, etc, at the same time. That is the case if our smoke alarm mechanisms work properly.

Specifically:

- create an element of the pathname space that means "all pids", say /proc/pids/
- reading it returns a series of data structures -- as many as will fit in the user's buffer
- the data returned is a stream of tagged data structures:

<data for pid1><data for tid1 of pid1><data for tid2 of pid1>....<data for pid2><data for tid1 of pid1> ...

- subsequent reads return more pid/tid data structures from where the last read() left off.
- each returned data structure is prefixed with a unique tag (since we will be mixing struct types in the same message

We want to generalize this to provide focused filtering and to allow customers to, well..., customize:

- define a set of tags. each tag maps to a struct type and a function for filling it in
- provide several tags for pids:
    - complete pid info
    - memory usage only
    - cpu usage only
    - etc
- provide several tags for tids:
    - full tid info
    - list of syncs only (for deadlock detection)
    - etc
- allot some tags to a customer, and allow them to define the fill-in functions in a customer-specific module.

The user specifies which set of structs to return with a special pathname: /proc/pids+threads+maps/ which means returns the thread data and map data for all pids in the system. Reading /proc/pids/ would return the full pid data for each pid in the system. This allows the user to read all possible data, or to filter data appropriate for a specific task.

Observation: this is equivalent to returning xml. For efficiency, we prefer to return packed binary. Later, we could easily add a layer (function or resmgr) to translate into xml.

Looks pretty slick to us.

More on A5: the Generic Notifier#

We would like to base soft RLIMIT warnings on a generic notifier. We'd also use it for all memory and aps notifications, time to do deadlock detection notifications, and in general any notification for any threshold transition of interest to any client of any resource in the OS. This means we need to come up with a pathname space to name all these possibliies:

- Colin will advise us on what pathname space to use for both the notifier and bulk reads of Proc data

Design Notes#

- we want user to be able to specify "notify me whenever any process opens more than 40 files" in addition to "notify me when process 4768 opens more than 15 files"
  - want the path name space to aggregate objects
  - specifying something like /proc/475730/notify_fdcount/ means notify on the fd count of only pid 475730.
  - specifying something like /proc/pids/notify_fdcount/ means notify when any process's fd count exceeds a limit
- a user specifies a resource, to be notified about, and a set of thresholds
- many users may ask for notifications about the same resource and set different thresholds
- one user may set several thresholds for the same resource (useful say for minor and major alarm levels)
- a threshold is a number and a direction. ex: "going above 40", or "dropping below 1"
  - the direction attribute of the threshold allows us to avoid flooding the user with notifications.
    - For example, if the threshold is FD's rising above 40, we will not send a second notification until the value drops to 39, or less, and then later rises to 40
- we should design a list of all resources we want to offer threshold detection for:
  - thread syncs, thread cpu, process memory, process FDs, system memory, system cpu time?, plus all rlimits

- We need to allow these operations in the api:
  1. set a threshold on a resource used by a specified pid or all pids (specfied tid or all tids etc)
     - returns a handle that identifies (owner_pid, resource_name, threshold value and direction)
     - handle should be <= 32bits
  2. read current usage of resource
  3. delete a threshold given the handle (optionally "delete all the thresholds that I've set")
  4. read thresholds for a given resource
     - may return several for the current user, and thresholds set by other users.
- we may wish to overwrite the pulse code in the sigevent, which we return to the client, to indicate which resource limit was hit. We'd write the resource handle into the sigevent. We would then expect the client to read the current resource value with that handle.

Implementation notes#

- To decide when to send a sigevent, we do not need to remember that we previously crossed the threshold. It is sufficent to emit a sigevent if (currant_usage < threshold) and (current_usage + amount_user_wants_now >= threshold). This assume we test the threshold every time we add to the current usage.

- An implementation would have a list of thresholds attached to each process entry. There would also be a list of global thresholds that apply to all processes. (This global threshold mechanism eliminates the need to generate 200 thresholds when one client says "notify me when any process does x".) On a resource modification, like opening a file, Proc would scan the list of global thresholds and the list of thresholds, attached to the process entry, to find all of the FDCOUNT thresholds. It would then issue sigevents for each that are triggered.

- For each threshold instance we need to store:
  - the id of the resource for which this is a threshold
  - the pid of the owner
  - the number of the limit
  - the direction of the limit
  - the sigevent to return the client

- a copy of of this threshold's handle
- a bit to say if we overwrite the user's sigevent with the resource handle