[Design Doc] File System Observer

This Document is public

Authors: dslee@chromium.org memmott@chromium.org (original author: asully@chromium.org)

Last Updated: Jul 30, 2024

One-page overview

Summary

The File System Access API lets web applications interact with the host operating system's file system. One big missing feature of it is allowing web applications to be notified when files or directories change. Web applications could try to implement something like this using polling, but that would be very inefficient. As such we want to provide a dedicated API for observing file changes at runtime while the site has an open tab.

Platforms

Mac, Windows, Linux, Chrome OS

Team

storage-dev@chromium.org

Bug

https://crbug.com/1019297

Code affected

//content/browser/file_system_access/
//third_party/blink/renderer/modules/file_system_access/
//third_party/blink/public/mojom/file_system_access/

Explainer

https://github.com/whatwg/fs/blob/main/proposals/FileSystemObserver.md

Motivation

The file system is a shared resource that can be modified from several contexts. A Bucket File System spans numerous agents - tabs, workers, etc - within the same storage key. The local file

system also spans across origins and other applications on the host operating system.

For a given agent to know about modifications to the file system - made either by itself or from some external context - it can currently poll the file system to detect changes. This is inefficient and does not scale well.

This doc proposes a FileSystemObserver interface which will much more easily allow a website to be notified of changes to the file system.

Observing file changes is *by far* the most-requested feature on the File System Access API from developers. See discussions on the spec: Watching/notifications · Issue #72

Javascript API Design

Web IDL

```
interface FileSystemObserver {
  constructor(FileSystemObserverCallback callback);
  Promise<void> observe(FileSystemHandle handle,
      optional FileSystemObserverObserveOptions options = {});
  void disconnect();
};

callback FileSystemObserverCallback = void (
    sequence<FileSystemChangeRecord> records,
    FileSystemObserver observer
);

enum FileSystemChangeType {
  "appeared",     // File/dir has been created or moved into the scope of the
                  // observation.
  "disappeared",  // File/dir has been deleted or moved out of the scope of the
                  // observation.
  "modified",     // File/dir has been modified.
  "moved",        // File/dir has been moved within the scope of the observation.
  "unknown",      // Zero or more events are missed. Site should poll the
                  // watched directory in response to this.
  "errored"       // This observation is no longer valid.
};

dictionary FileSystemObserverObserveOptions {
```

```
  bool recursive = false;
};

interface FileSystemChangeRecord {
  // The handle that was passed to FileSystemObserver.observe, representing
  // the scope of the observation.
  readonly attribute FileSystemHandle root;
  // The handle affected by the file system change. For “moved” type, this
  // refers to the new location of a moved handle.
  readonly attribute FileSystemHandle changedHandle;
  // The path of changedHandle relative to root.
  readonly attribute FrozenArray<USVString> relativePathComponents;
  // The type of change.
  readonly attribute FileSystemChangeType type;
  // Former location of a moved handle. Used only when type === ‘moved’.
  readonly attribute FrozenArray<USVString>? relativePathMovedFrom;
};
```

Observing Changes to a File

When the observed file changes, the website will receive a FileSystemChangeRecord including details about the file system change.

```
const callback = (records, observer) => {
  // Will be run when the observed file changes.

  for (const record of records) {
    // The change record includes a handle detailing which file has
    // changed, which corresponds to the observed handle in this case of
    // watching a file.
    const changedFileHandle = record.changedHandle;
    assert(await fileHandle.isSameEntry(changedFileHandle));

    // Since we're observing changes to a file, the root of the change
    // record also corresponds to the observed file.
    assert(await fileHandle.isSameEntry(record.root));

    handleRecord(record);
  }
}
```

```
const fileHandle = await window.showSaveFilePicker();
const observer = new FileSystemObserver(callback);
await observer.observe(fileHandle);
```

Observing Changes to a Directory

Non-recursive watching

```
const callback = (records, observer) => {
  for (const record of records) {
    assert(await directoryHandle.IsSameEntry(record.root));
    // Non-recursively watching a directory will only report changes to
    // immediate children of the observed directory.
    assert(record.relativePathComponents.length <= 1);

    handleRecord(record);
  }
}

const observer = new FileSystemObserver(callback);
const options = { recursive: false };  // Default is false.
await observer.observe(directoryHandle, options);
```

Recursive watching

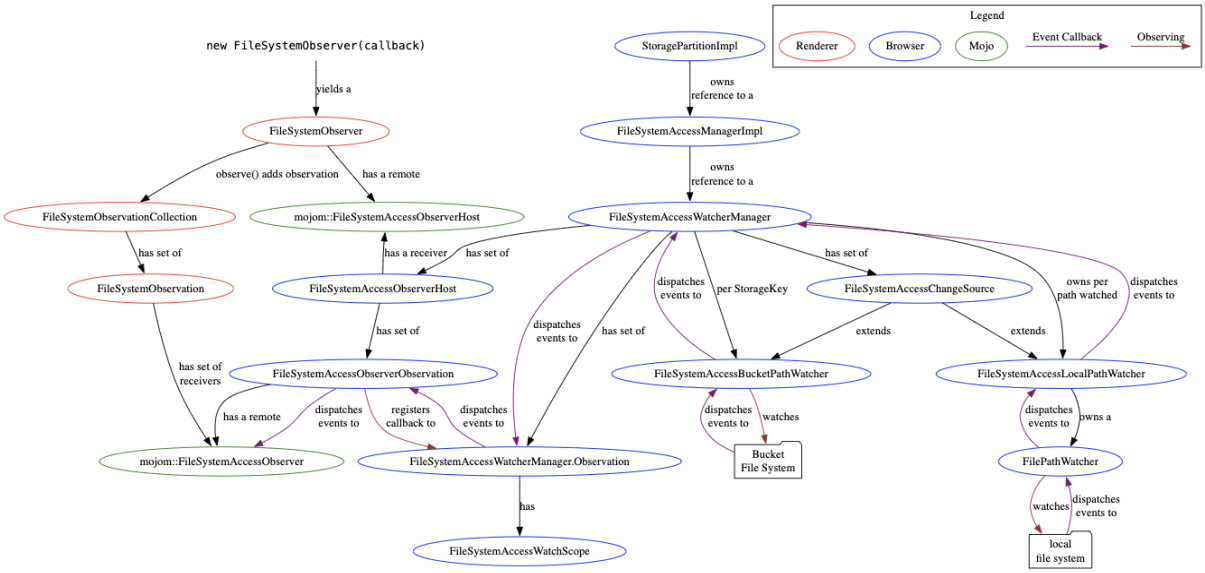
```
// Detecting changes to a directory recursively with a FileSystemObserver.
const callback = (records, observer) => {
  // Recursively watching a directory will report changes to both
  // children and all subdirectories of the watched directory.
  for (const record of records) {
    handleRecord(record);
  }
}

const observer = new FileSystemObserver(callback);
const options = { recursive: true };
await observer.observe(directoryHandle, options);
```

```
// Example function of handing FileSystemChangeRecord.
async function handleRecord(record) {
  // Decide how to mark the file dirty according to the
  // FileSystemChangeType included in each file system change record.
  switch (record.type) {
    case 'appeared':
      markCreated(record.root, record.relativePathComponents);
      break;
    case 'disappeared':
      // The relative path of the changed handle may be more useful than
      // the handle itself, since the file no longer exists.
      markDeleted(record.root, record.relativePathComponents);
      break;
    case 'modified':
      // A handle to the changed path may be more useful than its
      // relative path if reading from the file is necessary to
      // understand the change.
      //
      // Note that records with the 'modified' change type may be noisy
      // (e.g. overwriting file contents with the same data) so it's
      // necessary to check whether the file actually changed.
      if (await checkIfChanged(record.changedHandle)) {
        markModified(record.root, record.relativePathComponents);
      }
      break;
    case 'moved':
      // record.relativePathMovedFrom is used exclusively for 'moved'
      // records, to indicate the previous path of the moved file.
      markMoved(record.root, record.relativePathMovedFrom,
                record.relativePathComponents);
      break;
    case 'unknown':
      // Change occurred, but the type of change is unknown. Check
      // the root directory to search for changes.
      await searchForChange(record.root));
      break;
    case 'errored':
      // Watching paths on the local file system may fail unexpectedly.
      // After receiving a record with an 'errored' change type, we will
      // not receive any more change records from this observer.
      // You may then consider re-observing the handle, though that may
```

```
      // fail if the issue was not transient.
      observer.disconnect();
      break;
  }
  // ...
}
```

High-level Architecture



source [link]

[Renderer] FileSystemObserver

Renderer-side representation of FileSystemObserver javascript instance that holds mojo connection to the browser-side, via mojom::FileSystemAccessObserverHost, which handles observe() calls. Upon a successful observation from the browser-side, mojom::FilesystemAccessObserver, which is the receiver responsible for getting file change events notified from the browser-side, is returned to FileSystemObserver and added to FileSystemObservationCollection. This class is also responsible for checking the file system storage access.

[Browser] FileSystemAccessWatcherManager

This class manages all watches to file system changes for a StoragePartition. Raw changes from the underlying file system are plumbed through this class, to be filtered, batched, and transformed before being relayed to the appropriate Observer, represented by FilesystemAccessWatchManage::Observation.

FilesystemAccessWatchManager::Observation is responsible for relaying change events for a given FileSystemAccessWatchScope, to FileSystemAccessObserverObservation.

This class owns a set of FileSystemAccessChangeSource, which can be either FileSystemAccessBucketPathWatcher or FileSystemAccessLocalPathWatcher, based on which file system it watches. Currently, there is only one FileSystemAccessBucketPathWatcher, as it watches the root of the Bucket File System, while there may be many FileSystemAccessLocalPathWatchers.

[Browser] FileSystemAccessChangeSource

This class encapsulates the logic to watch a file system and to notify a file system change event, for a given FileSystemAccessWatchScope. Currently, it is extended by FileSystemAccessBucketPathWatcher and FileSystemAccessLocalPathWatcher.

[Browser] FileSystemAccessWatchScope

This class describes the extent of the file system that is being observed, which can be a single file, a directory and its contents, or a directory and all its subdirectories, or the whole file system (i.e. Bucket File System)

[Browser] FileSystemAccessObserverHost

This class stores the state associated with each FileSystemAccessObserverHost mojo connection, and interacts with FileSystemAccessWatcherManager to handle observe() calls. It also checks for a valid permission state before observe()ing.

[Browser] FileSystemAccessObserverObservation

This class is a browser-side representation of a successful FileSystemObserver.observe() call from Javascript. It registers a callback function to the corresponding FilesystemAccessWatchManager.Observation retrieved from FileSystemAccessWatcherManager. When this callback function is invoked, it checks for a valid permission state before forwarding changes to the observed file or directory to a mojo pipe, whose receiver is owned by the renderer.

[Browser] FilePathWatcher

This class is an encapsulation of platform-specific local file path watcher implementation, which calls an OS-specific API to watch local file system changes.

Setting up Observation: FileSystemObserver.observe()

On the renderer-side, FileSystemAccess.observe(FileSystemHandle handle) calls gets a FileSystemAccessTransferToken from the passed handle, and invokes mojom::FileSystemAccessObserverHost.observe() with the transfer token.

On the browser-side, FileSystemAccessObserverHost checks for a valid permission (i.e. granted) on the passed handle and gets a new FileSystemAccessWatcherManager::Observation from FileSystemAccessWatcherManager. If FileSystemAccessWatcherManager has an existing FileSystemAccessChangeSource that matches the scope, it will be re-used; otherwise, a new FileSystemAccessChangeSource is set up. Upon a successful retrieval of Observation, FileSystemAccessObserverHost creates a new FileSystemAccessObserverObservation, which is passed with (1) Observation (for receiving the changes) and (2) a newly created mojom::FileSystemAccessObserver (for reporting the changes back to the renderer).

Watching Local File System

Chromium has an existing base::FilePathWatcher class, which provides primitive watching of a given file path using file path watching APIs built-in to the host OS. It has a number of limitations, such as only reporting *that* a path has changed without any indication of *how*, as it is initially designed for this simple use case.

As the only (known) use case that needs the support for file change info, it was recommended to make a copy of base::FilePathWatcher as content::FilePathWatcher and to add new functionalities to the file system access code base (i.e. content/browser/file_system_access/file_path_watcher/...). We may consider consolidating two separate implementations, as the API shape evolves and more clients will need to be supported.

Interface Changes to FilePathWatcher

The following changes will be made to the interface, in order to report (1) change type and (2) modified file path(s).

```
using Callback = RepeatingCallback<void(ChangeInfo info, bool error)>;

enum class ChangeType {
  kUnknown,
```

```
  kCreated,
  kDeleted,
  kModified,    // Set if either attributes or contents are modified.
  kMoved,       // Set for any move (moved out of, into, or within the
                // watched scope).
};

// Path type of the modified path.
enum class FilePathType {
  kUnknown,
  kDirectory,
  kFile,
};

struct ChangeInfo {
  FilePathType file_path_type;
  ChangeType change_type;

  // Modified path of the changed file or directory.
  FilePath modified_path;

  // Previous path that the file or the directory has been moved from. This
  // field is only set for ChangeType:kMoved. It is provided as best-effort,
  // and may not exist if the underlying platform is unable to find information.
  std::optional<FilePath> moved_from_path;
};

bool Watch(const base::FilePath& path, const WatchOptions& options,
           const Callback& callback);
```

Platform-specific changes to FilePathWatcher

Windows

base::FilePathWatcher uses FindFirstChangeNotification which simply states that something has changed but doesn't provide the change type or path.

In the copied content::FilePathWatcher, we use ReadDirectoryChangesW which can recursively watch a directory and provides both the change type and path.

ReadDirectoryChangesW reports changes with a FILE_NOTIFY_INFORMATION struct. It's Action field can be mapped to content::FilePathWatcher::ChangeType as follows:

content::FilePathWatcher::ChangeType	FILE_NOTIFY_INFORMATION::Action
kCreated	FILE_ACTION_ADDED
kDeleted	FILE_ACTION_REMOVED
kModified	FILE_ACTION_MODIFIED
kMoved	FILE_ACTION_RENAMED_OLD_NAME FILE_ACTION_RENAMED_NEW_NAME

Windows sends move events as two consecutive FILE_ACTION_RENAMED_OLD_NAME and FILE_ACTION_RENAMED_NEW_NAME. These can be coalesced into a single kMoved event.

We ignore FILE_ACTION_MODIFIED on directories since it doesn't produce any useful information.

ReadDirectoryChangesW doesn't provide information on the change's file path type. This means we must fill out content::FilePathWatcher::ChangeInfo::file_path_type via base::GetFileInfo. This unfortunately leads to several issues:

- After receiving an event from the OS, there is a race between checking the event's file path type via base::GetFileInfo and further file events which may change the file path type.
- For delete events, there is no possible way to know the file path type with base::GetFileInfo since the file is already gone.
- Some FILE_ACTION_MODIFIEDs on directories may not get filtered due to the previous two issues.

This issue can be somewhat mitigated if we cached the known state of the file system. And it can further be mitigated if we used the file events to deduce the state of the file system. E.g. we know that if we receive an event for a file path, we know at that moment all its ancestors are directories and not files.

Deleting the directory that ReadDirectoryChangesW is watching can also cause problems. Some events before the watched directory is deleted can be lost. And the events between the directory being deleted and the ReadDirectoryChangesW being set up again will be lost.

Mac

Currently uses kqueues for non-recursive watches (and always on iOS) and FSEvents only for recursive watches on MacOS.

We could support non-recursive watches with FSEvents, if determined to be a worthwhile effort. Historically, kqueue support was added earlier and FSEvents was unreliable on Mac 10.6.

As a result, FSEvents was only used for recursive support that kqueue couldn't provide.

This comes with some downsides:

- MacOS and iOS are the only platforms for which:
  - watching a directory does not trigger events when children change, and
    - This would likely become an *enormous* headache for web developers
  - watching a symlink does not trigger events when the target of the symlink changes
- kqueue implementation is very noisy

FSEvents appears to have become much more reliable in Mac 10.7. From what I can tell, all the major file path watching third-party libraries (e.g.) use FSEvents from Mac 10.7 onwards.

Proposal: Use FSEvents for all* watches on MacOS.

ChangeEventInfo	FSEventStreamCallback (with FSEventStreamEventFlags)
FilePath	eventPaths
PathType	kFSEventStreamEventFlagItemIsDir + kFSEventStreamEventFlagItemIsFile + kFSEventStreamEventIs*Link
ChangeType::kUnknown	Unused
ChangeType::kModified	kFSEventStreamEventFlagItemModified kFSEventStreamEventFlagItemInodeMetaMod Also, probably: kFSEventStreamEventFlagItemChangeOwner And maybe or maybe not: kFSEventStreamEventFlagItemXattrMod + kFSEventStreamEventFlagItemFinderInfoMod
ChangeType::kCreate	kFSEventStreamEventFlagItemCreated Also, maybe: kFSEventStreamEventFlagItemCloned
ChangeType::kDeleted	kFSEventStreamEventFlagItemRemoved
ChangeType::kMoved	kFSEventStreamEventFlagItemRenamed

Linux / ChromeOS

Currently, it uses inotify. It already supports reporting modified file paths (unlike Mac and Windows), which are available from inotify_event struct sent from the inotify API. In order to report the change type, inotify_event.mask contains information about the type of change events, which will be passed to the callback registered to FilePathWatcher.

ChangeEventInfo	inotify_event
FilePath	name + len
PathType	mask.IN_ISDIR
ChangeType::kUnknown	Unused
ChangeType::kModified	mask.IN_ATTRIB + mask.IN_CLOSE_WRITE
ChangeType::kCreate	mask.IN_CREATE
ChangeType::kDelete	mask.IN_DELETE
ChangeType::kMoved	mask.IN_MOVED_FROM + mask.IN_MOVED_TO

On inotify, it generates two events for a move (IN_MOVED_FROM and IN_MOVED_TO) with a matching cookie value. The current implementation reports an event if the modified file path is within the scope of the watch root. This means that moving a child file within the watched directory (i.e. foo/bar -> foo/baz while watching foo/) would generate duplicate move events. The two related move events are coalesced as one, by matching the cookie value.

Known Issues

inotify

inotify has default max inotify watches per user is 8192. Since this is *per user*, this limit can be reached by other applications or by other usages within Chrome. If the limit is reached, an error would be sent to the client, and the site would need to re-try observe().

This limitation is exacerbated by the fact that an inotify watch is created per file component (and for each descendents, if watching recursively). This is a known limitation of inotify and a known problem in Chrome.

Mac (FSEvents)

A maximum of 4096 watched paths is allowed. If exceeded, calling start() would result in error.

Windows

A buffer is passed to Windows' ReadDirectoryChangesW for holding the file events. The buffer can be between 0 and 64kb. And whatever is passed, Windows will double the memory usage since it maintains its own copy. Having a buffer that is too small can lead to frequent buffer overflows. So there must be a balance between avoiding buffer overflows and reducing memory usage.

Move events between directories (vs. under the same directory) do not arrive as two consecutive FILE_ACTION_RENAMED_OLD_NAME and FILE_ACTION_RENAMED_NEW_NAME, making it not possible to coalesce related move events. It may be worth investigating if Windows NTFS change journal could help solve this inconsistency.

Watching Bucket File System

Unlike the local file system, OS-level file path watching APIs are not appropriate for the Bucket File System* (a.k.a. the OPFS):

- Its contents are not easily accessible by the user
- It has a flat directory structure
- It uses an in-memory file system in incognito mode

Since changes to the Bucket File System are coming from Chrome, we can add hooks on operations which modify files to track and report changes ourselves. This can be done by extending storage::FileChangeObserver in storage/browser/file_system/file_observers.h, which monitors changes to files managed by Chrome.

However, there are some known drawbacks:

- There is only one use of this class, which is in service of the deprecated SyncFileSystem for Chrome Apps
- FileSystemSyncAccessHandles do not use //storage/browser/file_system, and migrating FileSystemWritableFileStreams away from using FileSystemOperationRunner is on the table. Incorporating file modification changes from outside //storage/browser/file_system is doable, but would require adding additional hooks
- We would like to eventually move FSA away from using //storage/browser/file_system anyways.

FileSystemAccessBucketPathWatcher (a subclass of FileSystemAccessChangeSource for Bucket File System) will extend storage::FileChangeObserver for observing changes. It watches the root of the Bucket File System (which is a flat structure on disk), so there is only

one instance of FileSystemAccessBucketPathWatcher for FileSystemAccessWatcherManager.

Processing and Reporting File Changes

Raw changes (ChangeInfo) notified to FileSystemAccessChangeSource from different file system backends are routed to its observer, FileSystemAccessWatcherManager. These raw changes are then processed (i.e. handling errors, converting certain move events, etc) and sent to the Observations with the matching scope.

Each Observation's registered callback (to FileSystemAccessObserverObservation) is invoked. Further processing is done in FileSystemAccessObserverObservation, such as checking for valid permission and active RenderFrameHost. See below for more details.

Reporting "move in/out of" the scope as "appeared/disappeared"

Move/rename could occur across the boundary of the observation scope. If a file/directory is moved into the observation scope, then the event is reported as "appeared". Similarly, if it is moved out of the observation scope, the event is reported as "disappeared". Otherwise, it is moved within the observation scope, and it is reported as "moved" with both old and new path information. This ensures that path information is not exposed if it is out of the observation scope, with the assumption that the site does not have permission to the file system outside the observation scope. Furthermore, it helps mitigate the platform inconsistency on Windows where some move events are reported as created or deleted.

Handling OS-level Error

An error can occur on watching on inotify if the number of inotify watches exceeds the default maximum value. In this case, the callback is reported with "errored" type, and the observation is destroyed. The site is expected to handle the error, by re-attempting to observe() again.

Handling Permission Change

A file or directory path can be watched only if a corresponding FileSystemHandle has "granted" permission status. In addition to checking the permission status (1) when an observation is created and (2) before FileSystemObserver callback is invoked, permission change is observed in FileSystemAccessObserverObservation by making it FileSystemAccessPermissionGrant::Observer. In the case of permission status change to values other than "granted", the callback is invoked with "errored" and the observation is destroyed.

Interactions with BFCache

Since no events are sent to the callback while the page is not active, some change events may be lost if the page was in BFCache and then back to “active” state. In this case, we would track whether a file change event has occurred while in BFCache, and send an “unknown” event to the callback once the page is back to “active”. This is tracked via WebContentsObserver::RenderFrameHostStateChanged.

In order to avoid exposing what exactly occurred (i.e. exact sequence of events) while in BFCache, a single change event with “unknown” type is sent such that the site can know that *some* change occurred.

Ignoring Swap Files

Using the File System Access API, FileSystemWritableFileStream can be created via FileSystemFileHandle.createWritable(). This creates a swap file (.crswap), where temporary writes are recorded until it overwrites the original file. This can cause additional file change events to be emitted. Since this is an implementation detail of FileSystemWritableFileStream, any changes to swap files will be ignored such that only changes in the original file would be reported.

FileSystemObserver.disconnect()

Disconnecting the observer destroys all observations set up under this observer, and cleans up watches (unless there are other observations using the watches).

One drawback of disconnect() is that it disconnects “all” observations. In the future iteration, we may consider adding FileSystemObserver.unobserve(FileSystemHandle), that would un-observe individual observation.

Resource Consumption Considerations

Limit on the number of inotify watches

As mentioned above, watching can fail on inotify if the global limit is reached. We could consider imposing a limit from FilePathWatcher, as to prevent the starvation outside FilePathWatcher use in Chrome. However, estimating an upper limit is challenging as Chrome does not have a visibility on how inotify is used by other applications, and the usage pattern will vary widely across users. Also, the default max could be changed by user, which would make imposing a fixed upper limit infeasible.

During the Origin Trial, the resource exceeded error will be monitored in order to understand how often this occurs, and whether/what limit should be imposed (at Chrome level, or at File

System Observer API level) will be revisited.

Note that this estimated usage would not include the usage occurring outside of Chrome by other applications, nor the usage within Chrome that does not use FilePathWatcher as a layer. In other words, this restriction would only help to prevent FilePathWatcher consuming all the inotify watches.

Limit on the number of observations per origin

Similarly, we may consider adding a per-origin (or tab) limit in order to prevent a site hoarding the OS resource. The challenge is that the number of inotify watches created is not equal to the number of FilePathWatcher::Watch*() calls, since an inotify watch is created per file component. Also, the number of FilePathWatcher::Watch*() calls is not equal to the number observations created (via observe() Javascript API), since a same watch could be shared by multiple observations, if the configuration is the same (= overlapping observations).

To come up with the estimated usage that is closest with the actual OS resource usage, the number of (inotify) watches should be counted, if possible.

During the Origin Trial, the number of observations created will be recorded in order to learn about the usage pattern and to determine this limit.

Followup Work

FileSystemObserver.unobserve()

We may consider adding unobserve() to the API, which will allow disconnecting individual observation. This would be particularly useful when handling an “errored” change event in which the observation is already destroyed; instead of disconnecting the whole FileSystemObserver, it can simply unobserve a single observation.

Optimize FilePathWatcher

- On inotify, ref-counting inotify watches to reduce open watch descriptors (and possibly allow the use of IN_MASK_CREATE).
- Reduce calls to UpdateWatch[es]()
    - On inotify, UpdateWatches() is called every time a component of the watched path (dis)appears, whereas ideally it would only be called for directory changes and symlinks
    - On Windows, UpdateWatch() + StartWatchingOnce() is called for every change. This is wasteful, since the watch really only should be updated if the path of the watched directory itself changes, as we do on some other platforms.

Optimize Overlapping Observations

It is possible that there may be overlapping observation scopes (within the same site, or across different sites). Currently, if there is a watch set up for a given scope, it is re-used. However, if the scopes are not exactly the same, it is not re-used. The handling of overlapping observations could be improved for re-use.

Batching multiple change events at once to JS API

The site may want to receive events at a batch, instead of receiving it right when the change occurs. It may be able to specify a time window for which a batch of change events are sent.

Filtering Options

The site may want to specify options to include or exclude certain change types. This would help reduce the number of callbacks for the events that the site is not interested in.

Metrics

UMA

- Javascript API usage will be tracked with UseCounter
- OS watch API error (i.e. resource limit exceeded)
- OS watch API callback error
- Rate at which the number of callbacks sent for an observation for a one-second window

UKM

FileSystemObserver.observe() is tracked for UKM logging for the Origin Trial, given a small set of origins using this API.

Rollout plan

This API will be initially available via Origin Trial from M129 to M134.

Testing plan

Web Platform Test

Manual wpt will be enabled for the File System Observer feature. (Automatic testing disabled

per b/346991169)