The problem reproduces with a specific sequence involving three threads.

1) sun.awt.AWTAutoShutdown.theInstance.blockerThread
2) a client thread posting an event using EventQueue.postEvent
3) the current java.awt.EventQueue.dispatchThread (java.awt.EventQueue.dispatchThread)

The problem occurs when the client thread posts the event **after** the shutdown thread posts its event to shutdown, and **before** the current dispatch threads completes its shutdown processing.

Initial state:

The environment is headless (there are no UI components, no peer events, only java events).
The latest java event has been processed a little over a second ago.

1a) **AWTAutoShutdownThread**.
After AWTAutoShutdownThread.SAFETY_TIMEOUT expires (> 1 second without events), this thread calls AppContext.stopEventDispatchThreads.
That call results in posting an event to shut down the dispatch thread.
EventQueue.pushPopLock is being held while the event is posted, so other threads must wait until that lock can be obtained.

1b) **Client thread.**
The client thread wants to post an event, but must wait for EventQueue.pushPopLock

1c) **EventDispatchThread.**
Meanwhile, the current dispatch thread can pick up this event, and dispatch it. (see code below)
This sets EventDispatchThread.doDispatch to false, and the thread's main loop exits.

```
private void dispatchEventImpl(final AWTEvent event, final Object src) {
    event.isPosted = true;
    if (event instanceof ActiveEvent) {
        // This could become the sole method of dispatching in time.
        setCurrentEventAndMostRecentTimeImpl(event);
        ((ActiveEvent)event).dispatch();
    } else if (src instanceof Component) {
        ((Component)src).dispatchEvent(event);
        event.dispatched();
    } else if (src instanceof MenuComponent) {
        ((MenuComponent)src).dispatchEvent(event);
    } else if (src instanceof TrayIcon) {
        ((TrayIcon)src).dispatchEvent(event);
    } else if (src instanceof AWTAutoShutdown) {
        if (noEvents()) {
            dispatchThread.stopDispatching();
        }
    } else {
        if (eventLog.isLoggable(PlatformLogger.FINE)) {
            eventLog.fine("Unable to dispatch event: " + event);
        }
    }
}
```

The **EventDispatchThread** now needs to execute its finally block. (see code below)

Note that doDispatch is set to false, so detachDispatchThread is called with forceShutdown == true.

Note that EventQueue.detachDispatchThread requires EventQueue.pushPopLock, so both the client thread and the current dispatch thread are now competing for this lock.

```
public void run() {
    while (true) {
        try {
            pumpEvents(new Conditional() {
                public boolean evaluate() {
                    return true;
                }
            });
        } finally {
            // 7189350: doDispatch is reset from stopDispatching(),
            //    on InterruptedException, or ThreadDeath. Either way,
            //    this indicates that we must force shutting down.
            if (getEventQueue().detachDispatchThread(this,
                        !doDispatch || isInterrupted()))
            {
                break;
            }
        }
    }
}
```

Processing new client event:

2a) The **client thread** obtains EventQueue.pushPopLock **before** the dispatch thread can obtain it in detachDispatchThread.

While the client thread's event has the lock, EventQueue.dispatchThread is checked for null. (see code below). It's not null, so no new dispatch thread gets created.

2b) The **dispatch thread** has to wait until the client event is posted.

```
private final void postEventPrivate(AWTEvent theEvent) {
    theEvent.isPosted = true;
    pushPopLock.lock();
    try {
        if (nextQueue != null) {
            // Forward the event to the top of EventQueue stack
            nextQueue.postEventPrivate(theEvent);
            return;
        }
        if (dispatchThread == null) {
            if (theEvent.getSource() == AWTAutoShutdown.getInstance()) {
                return;
            } else {
                initDispatchThread();
            }
        }else{
        }
        postEvent(theEvent, getPriority(theEvent));
    } finally {
        pushPopLock.unlock();
    }
}
```

Completing detachDispatchThread

The current event **dispatch thread** executes detachDispatchThread.
Since it's called with forceDetach == true, it will **not** execute peekEvent() and it will shut down.

```
final boolean detachDispatchThread(EventDispatchThread edt, boolean forceDetach) {
    /*
     * This synchronized block is to secure that the event dispatch
     * thread won't die in the middle of posting a new event to the
     * associated event queue. It is important because we notify
     * that the event dispatch thread is busy after posting a new event
     * to its queue, so the EventQueue.dispatchThread reference must
     * be valid at that point.
     */
    pushPopLock.lock();
    try {
        if (edt == dispatchThread) {
            /*
             * Don't detach the thread if any events are pending. Not
             * sure if it's a possible scenario, though.
             *
             * Fix for 4648733. Check both the associated java event
             * queue and the PostEventQueue.
             */
            if (!forceDetach && (peekEvent() != null) || !SunToolkit.isPostEventQueueEmpty())

                return false;
            }
            dispatchThread = null;
        }
        AWTAutoShutdown.getInstance().notifyThreadFree(edt);
        return true;
    } finally {
        pushPopLock.unlock();
    }
}
```

Final state:

EventQueue.dispatchThread is null, and the queue contains one event.