Technical Note TN2050

Observing Process Lifetimes Without Polling

CONTENTS

This technote shows how to determine the process lifetime of running processes, without polling the process list. This is done through four different methods: Carbon Event notifications, Cocoa NSWorkspace and Cocoa-Java NSWorkspace notifications and last through Mach port dead-name notifications. The technote describes each method in detail and provides full examples of getting notifications through each mechanism.

Carbon developers will likely want to use the Carbon Events solution, and Cocoa developers the NSNotificationCenter solution and Java developers would likely want to use the Cocoa-Java NSNotificationCenter notifications. The Mach port solution can be used by all developers including pure UNIX developers.

The Carbon, Java and Cocoa solutions only provide notifications for processes which use the Carbon or Cocoa frameworks in some way. Pure UNIX processes will not register on the Carbon and Cocoa notifications. The Mach port dead-name can notify on any running process including pure UNIX processes. However, registration for the Mach port notification must be done on a per-process basis and can only be done for processes already running.

This technote is directed at any developers who make use of the process list.

[Jul 01 2002]

Introduction

Determining when certain processes are launched and terminated has long been a problem developers have been trying to

solve. In the past developers were forced to poll the process list in order to determine if certain processes were running or not. Polling the process list is a very costly solution to this problem. Developers no longer have to poll the process list to determine if processes are running or not. This is because they can register for notification of process launch/termination through a variety of methods. Various methods of getting notification of process launch/termination are discussed in this technote.

How the Notifications Work

Carbon, Java and Cocoa Notifications

For the high-level notifications (via Carbon, Java and Cocoa) on Mac OS X whenever an application is launched or terminated it needs to connect to the window server. When this connection occurs the window server takes note of the connection and notifies any interested parties that the connection occurred. In this case the Carbon and Cocoa frameworks receive the notifications and pass them along to any interested parties. Similarly, whenever an application terminates the windowserver notices the disconnection and sends out a termination notification to the frameworks. Again these notifications eventually propagate to interested applications.

Mach Port Notifications

Every Unix process, from a full-scale application to a simple Unix tool, has a Mach task associated with it. Each Mach task is represented by a Mach port. Messages directed at that port allow the process state to be inspected and controlled. When the process exits, the associated Mach task is terminated, and the Mach port associated with that task is invalidated (destroyed). Any other processes which holds a send-right to that (now "dead") task port can detect this transition, and therefore detect process exits, by registering for a Mach port notification.

Carbon Events Notifications

You can determine process lifetime information using a variety of methods. One approach (which doesn't require polling) is the use Carbon Events. By registering for the Carbon Events `kEventAppLaunched` and `kEventAppTerminated` you can receive notification of process launches or terminations. This approach is demonstrated in the code listing below. Note that the code works on both Mac OS 9 (CarbonLib 1.3.1 and later) and Mac OS X (10.0 and later).

The function you will be interested in code listing is the `InstallLaunchTerminationNotifierFunctions()` function. This function takes function pointers which represent the functions which get called when a launch or termination occurs. The code uses the generic functions `LaunchFunction()` and `TerminateFunction()` to receive the callbacks. However, you will likely want to change or replace these generic functions with your own functions. The `InstallLaunchTerminationNotifierFunctions()` call will add the callbacks to the current ApplicationEventLoop. Note that the ApplicationEventLoop must be run in order for the callback functions to be called properly. Note the `InstallLaunchTerminationNotifierFunctions()` function is re-enterant.

Note the Carbon notifications will only fire for processes which require a connection to the window server. That is processes which use the higher-level frameworks in some way. This means that certain types of processes (for example, low-level UNIX processes) won't cause launch/termination notifications.

```
#include <Carbon/Carbon.h>

typedef void (*LaunchNotificationCallBack)
                    (ProcessSerialNumber* LaunchedApplicationPSN,
                        CFStringRef LaunchedApplicationProcessName,
                        CFURLRef CFURLForLaunchedApp);

typedef void (*TerminationNotificationCallBack)
                    (ProcessSerialNumber* TerminatedApplicationPSN);

struct LaunchTerminationCallbackInfoStructure
{
    LaunchNotificationCallBack launchCallback;
    TerminationNotificationCallBack terminationCallback;
};

/*****************************************************
 * LaunchFunction
 *****************************************************
 * Purpose:  This is the callback function which gets called any
 * time a process gets launched on the system.   This is because it
 * was the callback function specified when calling
 * InstallLaunchTerminationNotifierFunctions.  Note that since
 * we are talking about processes we mean "normal" processes.
```

```
 * This excludes processes which are launched which are daemons
 * or low-level UNIX type processes.  Note that before this
 * function will be called InstallLaunchTerminationNotifierFunctions
 * must be called.  Also, the CarbonEvents event loop must be run
 * before the callback functions will be properly called.
 *
 * Parameters:
 *
 *  LaunchedApplicationPSN      A Process Serial Number
 * of the process which was just launched.   When
 * LaunchFunction is called this parameter will have a value
 * representing the PSN of the process which caused the notification.
 * Note if there was an error getting the PSN then NULL will be passed
 * for this parameter.
 *
 *  LaunchedApplicationProcessName      A CFString containing
 * the process name of the process which was just launched.
 * When  LaunchFunction is called this parameter will have a value
 * representing the process of the process just launched.  Note if there
 * was an error looking up the process name than this parameter will be
 * null.
 *
 *  PathToLaunchedApp       A CFURL pointing to the location
 * of the process which was just launched.   When  LaunchFunction is called
 * this parameter will have a value representing the location where the process
 * was launched from.  Note if there was an error looking up the path to the
 * launched process than this parameter will be NULL.
 *
 ************************************************/

void LaunchFunction(ProcessSerialNumber* LaunchedApplicationPSN,
                    CFStringRef LaunchedApplicationProcessName,
                    CFURLRef PathToLaunchedApp)
{
    /* If the PSN, CFURL or process name passed was NULL this means
     * proxy callback function could not get all the information.
     * Here we just print out message and return
     */
    if ((LaunchedApplicationPSN == NULL)
    || (LaunchedApplicationProcessName == NULL)
    || (PathToLaunchedApp == NULL))
    {
        printf("Launch callback was called but missing"
                "some extra information on process\n");

        if (LaunchedApplicationPSN != NULL)
        {
            printf("PSN = hi: %ld, low: %ld\n\n",
                  LaunchedApplicationPSN->highLongOfPSN,
                  LaunchedApplicationPSN->lowLongOfPSN);
        }

        fflush(stdout);
    }
    else
    {
        /* Got notification with PSN, process name and Path to app.
         We will print out the application PSN and the process name.
         */
        printf("Application Launched: "); fflush(stdout);
        CFShow(LaunchedApplicationProcessName);
        printf("PSN = hi: %ld, low: %ld\n\n",
                            LaunchedApplicationPSN->highLongOfPSN,
                            LaunchedApplicationPSN->lowLongOfPSN);
```

```
        fflush(stdout);
    }
}

/*************************************************
 * TerminateFunction
 *************************************************
 * Purpose:  This is the callback function which gets called any
 * time a process gets terminated on the system.   This is
 * because this function was specified as a callback function in the
 * InstallLaunchTerminationNotifierFunctions call.  Note that since
 * we are talking about processes we mean "normal" processes.
 * This excludes processes which are terminated which are daemons
 * or low-level UNIX type processes.  Note that before this
 * function will be called InstallLaunchTerminationNotifierFunctions
 * must be called.  Also, the CarbonEvents event loop must be run
 * before the callback functions will be properly called.
 *
 * Parameters:
 *
 *  TerminatedApplicationPSN        A Process Serial Number
 * of the process which was just terminated.   When
 * TerminateFunction is called this parameter will have a value
 * representing the PSN of the process which caused the notification.
 * Note if there was an error getting the PSN then NULL will be passed
 * for this parameter.
 *
 *************************************************/
void TerminateFunction(ProcessSerialNumber* TerminatedApplicationPSN)
{
    /* If the PSN passed was NULL this means proxy callback function
     * couldn't get the PSN for some reason.  In this case we just
     * print out message and return
     */
    if (TerminatedApplicationPSN == NULL)
    {
        printf("Terminate callback was called but missing PSN\n");
        fflush(stdout);
    }
    else
    {
        /* Got the PSN so list the PSN of the application
        that terminated
         */
        printf("Application Terminated\n");
        fflush(stdout);
        printf("PSN = hi: %ld, low: %ld\n\n",
                    TerminatedApplicationPSN->highLongOfPSN,
                    TerminatedApplicationPSN->lowLongOfPSN);
        fflush(stdout);
    }
}

/*************************************************
 * LaunchTerminationNotifierProxyCallbackFunction
 *************************************************
 * Purpose:  This callback function will be called on launch
 * or termination of a process.  This function will in turn
 * call the launch and terminate callback functions which were
 * specified in the call InstallLaunchTerminateNotifiers.  In the
 * case of the launch notification function it will also pass the PSN
 * process name and path to the process of the process which just
 * launched.  In the case of the termination notification function
 * this function will pass the PSN of the process which just terminated.
```

```
 * Note this function will only be called after the
 * InstallLaunchTerminateNotifiers has been called and the associated
 * Carbon Event loop is run.
 *
 * Parameters: (All parameters are required if this function is to
 * be called as a Carbon Events
 * callback function)
 *
 *  theEvent    An event reference.  When
 * When  LaunchTerminationNotifierProxyCallbackFunction gets called this
 * variable will contain the actual event which caused the notification.
 * Since we are only registered for launch and termination events we
 * will only receive these types of events.  Inside the passed event is the
 * type of event (launch or termination) as well as the PSN of the
 * process which caused the notification.
 *
 *  userData        A generic pointer pointing to data associated
 * with the callback.  We use this generic pointer to store function
 * pointers to the launch and termination functions.
 ************************************************/
pascal OSStatus LaunchTerminationNotifierProxyCallbackFunction(
                EventHandlerCallRef nextHandler,
                EventRef theEvent, void* userData)
{
    OSStatus result;
    UInt32 eventKind;
    ProcessSerialNumber launchedTerminatedProgramPSN;
    ProcessSerialNumber* pSNPointerForCallbackFunction;
    EventParamType actualTypeReturned;
    UInt32 actualSizeOfDataReturned;
    CFURLRef cFURLForLaunchedTerminatedProcess;
    CFStringRef processNameOfLaunchedTerminatedApp;
    struct LaunchTerminationCallbackInfoStructure* LaunchTerminationStructure;
    LaunchNotificationCallBack launchCallback;
    TerminationNotificationCallBack terminationCallback;

    /* We have been passed the actual event which caused the callback.
     * Whether the event was a launch event or a termination event we
     * still want to know what process caused the event.   We can get
     * the process serial number of the process who caused the
     * notification.  This is done by pulling the parameter
     * from the event passed in.  We get the process serial number
     * from the event using the call GetEventParameter
     * First Argument: The actual event which contains the
     *                 processSerialNumber.
     *    In this case we pass the event which was passed to this
     *    callback function.
     * Second Argument: The 'name' of the parameter we want to get
     *    from the event.   this case we want to get the
     *    kEventParamProcessID parameter.  This is because it
     *    contains the process serial number we desire.
     * Third Argument: The type of the parameter we wish to get.
     *    In this case it is a process serial number as we want
     *    to get the serial number of the process that caused
     *    the callback.
     * Forth Argument: On return from the GetEventParameter call
     *    this parameter contains the type returned.  This type
     *    needs to match what is expected (typeProcessSerialNumber).
     *    We check this result to ensure its correct so pass a
     *    variable which can hold this result.
     * Fifth Argument: The size of the variable used to store
     *    the data returned. In this case the variable
     *    LaunchedTerminatedProgramPSN is used to hold the
     *    result PSN.  Thus, for size of output buffer we pass
```

```
     *    sizeof(LaunchedTerminatedProgramPSN).
     * Sixth Argument: On return from GetEventParameter this
     *    parameter will have the actual size of the data returned.
     *    We store this result to check to see this value is
     *    non-zero later.
     * Seventh Argument: Finally this is the variable on return
     *    will hold the Process serial number of the process which
     *    caused to the notification.
     * Return Value: A error result indicating result of the
     *    function.  If this value is zero then function was successful.
     */
    result = GetEventParameter(theEvent,kEventParamProcessID,
                typeProcessSerialNumber,&actualTypeReturned,
                sizeof(launchedTerminatedProgramPSN),
                &actualSizeOfDataReturned,
                &launchedTerminatedProgramPSN);

    /* Need to check PSN returned to ensure it is valid.  If the PSN
     * is valid we will proceed to lookup the location of the process
     * based on PSN.  Then afterwards pass both results to the
     * appropriate callback function. Alternatively, if the PSN is
     * invalid then pass NULL for PSN and location of process to the
     * callback functions.  Note in either case the Launched Terminated
     * callback functions get called since an event did occur.
     */
    if ((result == noErr)
    && (actualSizeOfDataReturned > 0)
    && (actualTypeReturned == typeProcessSerialNumber))
    {
        FSRef fSRefForLaunchedTerminatedProcess;

        //Setting the PSN which will be passed to the termination callback.
        pSNPointerForCallbackFunction = &launchedTerminatedProgramPSN;

        // --- Getting CFURL representing process launched/terminated --- //

        /* Similar to above we will now use the process serial number
         * to do a lookup of the process location.  We do this lookup
         * using a call to GetProcessBundleLocation.  Note that this
         * lookup only works on running applications.  Thus, if this
         * is a termination callback this function call will fail.
         * First Argument: The PSN of the process we are looking up.
         *     In this case we use the process serial number we just
         *     found.
         * Second Argument: The FSRef where the result will be placed
         *     on success.
         * Return Value: A system error indicating success or error
         *     on the system.  If the value is zero (noErr) this
         *     indicates success.
         */
        result = GetProcessBundleLocation(&launchedTerminatedProgramPSN,
                            &fSRefForLaunchedTerminatedProcess);

        if (result == noErr)
        {

            /* If no error making FSRef then we continue to get the CFURL.
             * Now we create the CFURL from the FSRef.  We do the creation
             * using the call CFURLCreateFromFSRef.  This is the CFURL
             * we will pass to the launch, terminate callback function.
             * Note on failure the CFURL will be null which is what we
             * want anyways on failure.
             * First Argument: Allocator to use.  We want default allocator
             *    as usual so pass null.
```

```
            * Second Argument: The FSRef to be used as a basis for creating
            *    the CFURL.
            */
           cFURLForLaunchedTerminatedProcess =
                                CFURLCreateFromFSRef(NULL,
                                &fSRefForLaunchedTerminatedProcess);
       }
       else
       {
           /* in case of error we want CFURL passed to callback function
            *  to be null. Note that the CFURL will always be null on a
            *  Termination notification. This is because you can't lookup
            *  the FSRef using the process manager
            */
           cFURLForLaunchedTerminatedProcess = NULL;
       }

       // --- Getting CFString representing process name of the --- //
       // --- launched/terminated application --- //

       /* Now we just got the process serial number we can do a lookup of the
        * process name.  We do this lookup using a call to CopyProcessName.
        * Note you can only get process name of running applications.
        * Thus if this is a termination callback getting the process name will
        * be unsuccessful (we don't use result anyways).
        * First Argument: The PSN of the process we are looking up.  In this
        * case we use the process serial number we just found.
        * Second Argument: The CFString where the process name will be placed
        * on successful result.
        * Return Value: A system error indicating success or error on the system.
        * If the value is zero (noErr) this indicates success.
        * Note that you can't lookup the process name if a process isn't running
        * (termination notification).  In that case we would see an error result.
        */
       result = CopyProcessName(&launchedTerminatedProgramPSN,
                                &processNameOfLaunchedTerminatedApp);
       if (result != noErr)
       {
           /* in case of error we want CFString passed to callback
            * function to be null.
            */
           processNameOfLaunchedTerminatedApp = NULL;
       }
   }
   else
   {
       /* Were not able to get the PSN thus we will pass null for all
        * parameters to the callback functions. This way the callback
        * functions still get called even though information is missing.
        * Note this shouldn't happen but would be the case if the event
        * passed in had a malformed PSN.
        */
       pSNPointerForCallbackFunction = NULL;
       processNameOfLaunchedTerminatedApp = NULL;
       cFURLForLaunchedTerminatedProcess = NULL;
   }

   /* --- Getting Launch Termination callback functions from
    * callback userData ---
    */

   /* Getting the launch termination structure from the generic pointer passed
    * to this callback function.  The generic pointer in this case was set
    * by us when we installed the notifications.  The generic pointer in this
```

```
     * case holds the function pointers for the launch and termination functions
     */
    LaunchTerminationStructure =
                    (struct LaunchTerminationCallbackInfoStructure*) userData;

    //Getting the launch and termination functions from
    //the structure passed to this function.
    terminationCallback = LaunchTerminationStructure->terminationCallback;
    launchCallback = LaunchTerminationStructure->launchCallback;

    /* Getting the event kind of the event that caused the callback function
     * to be called.  In our case the kind will either be kEventAppLaunched
     * or kEventAppTerminated.  If the kind is kEventAppLaunched we of course
     * call the launch callback function specified earlier.  Similarly if the
     * kind is kEventAppTerminated we will eventually call the terminate
     * callback notification function.
     * We get the event kind will a call to GetEventKind().
     * First Argument: The actual EventRef describing the event that occurred.
     *     This is the event we are getting the kind of.
     * Return Value: The event kind.  In this case the result will
     *     either be kEventAppTerminated or kEventAppLaunched.
     */
    eventKind = GetEventKind(theEvent);

    /* Calling the appropriate callback function based on the event which
     * was passed to us.
     * Also passing the PSN pointer we gathered earlier.
     * Note the event kind should either be
     * kEventAppTerminated or kEventAppLaunched it isn't so something
     * went wrong.  In the case where its not one of expected events the
     * callbacks are not called since a matching event did not occur.
     */
    if (eventKind == kEventAppLaunched)
    {
        /* calling launch callback function specified.  Note if there was
         * an error in  lookup of any of the properties needed then null
         * will be passed to the callback function
         */
        launchCallback(pSNPointerForCallbackFunction,
                       processNameOfLaunchedTerminatedApp,
                       cFURLForLaunchedTerminatedProcess);
    }
    else if (eventKind == kEventAppTerminated)
    {
        /* Calling termination callback specified.  Note that we don't
         * pass the process name or CFURL if its a termination event.
         * Reason is because the lookup of this information isn't
         * possible via PSN if the process is no longer running.
         */
        terminationCallback(pSNPointerForCallbackFunction);
    }

    //Releasing CFString which was allocated earlier.
    if (processNameOfLaunchedTerminatedApp != NULL)
    {
        CFRelease(processNameOfLaunchedTerminatedApp);
    }

    //Releasing CFURL which was allocated earlier.
    if (cFURLForLaunchedTerminatedProcess != NULL)
    {
        CFRelease(cFURLForLaunchedTerminatedProcess);
    }
```

```
    /* When we return from a carbon event callback function
     * we always want to be sure to call the next handler.
     * we do this with a call to CallNextEventHandler.
     * First Argument: The next event handler.  In this case the
     *    handler which was passed to this function
     * Second Argument: The event to pass along.  In this case
     *    the same event which was sent to us.
     */
    return(CallNextEventHandler(nextHandler, theEvent));
}

/*************************************************
 * InstallLaunchTerminationNotifierFunctions
 *************************************************
 * Purpose:  This function will install the given Launch/Terminate
 * notification functions.  On successful install the launch notification
 * function will be called any time a process is launched.  Also, the
 * termination notification will be called any time a process terminates.
 * Note that the notifications only apply to 'regular' processes.  They
 * don't apply to daemon or background applications.
 *
 * Parameters:
 *  YourLaunchNotificationFunction      A function pointer to a
 * void function with arguments: ProcessSerialNumber, CFStringRef, CFURLRef.
 * Before calling InstallLaunchTerminationNotifierFunctions the
 * YourLaunchNotificationFunction parameter is expected to have a function
 * which the user desires to be called any time a process launches.
 * Note that when called the parameters will have the following values:
 * ProcessSerialNumber argument will have the ProcessSerialNumber of the
 * process which just launched, the CFString will have the process name
 * of the process just launched and the CFURL will point to the application
 * which just launched.  After calling this routine the launch
 * notification function will be called any time a process launch occurs.
 *
 *  YourTerminationNotificationFunction     A function pointer
 * to a void function with a ProcessSerialNumber argument.
 * Before calling InstallLaunchTerminationNotifierFunctions the
 * YourTerminationNotificationFunction parameter is expected to have a
 * function which the user desires to be called any time a process
 * terminates.  Note that when called the ProcessSerialNumber argument
 * will have the ProcessSerialNumber of the process which just terminated.
 * After calling this routine the launch notification function will be
 * called any time a process launch occurs.
 *
 *  ErrorReturnValue        An OSStatus variable.  Upon
 * calling InstallLaunchTerminationNotifierFunctions this variable should
 * point to a pre-allocated OSStatus variable.  On return from
 * InstallLaunchTerminationNotifierFunctions this function will have a
 * system error result of the function.  A system error result will be as
 * defined in MacErrors.h.  Note that if the function return
 * value is zero then the ErrorReturnValue will also be zero (noErr)
 *
 *  EventHandlerRemovalHandle   A pointer to a EventHandlerRef.
 * When calling InstallLaunchTerminationNotifierFunctions this variable
 * should point to a pre-allocated EventHandlerRef.  On return from
 * InstallLaunchTerminationNotifierFunctions this function will hold
 * an EventHandlerRef which can be used to remove the installed
 * Carbon Event handler.  This is done by calling the
 * RemoveLaunchTerminationNotifierFunctions giving it the returned handler.
 *
 *  <Function Result>     A integer return value.
 *              See result codes listed below.
 *      Result Codes:
 *      -4  Unable to install launch/termination event handler
```

```
 *      -3  Error getting UPP for callback function
 *      -2  Error allocating memory for callback function information
 *      -1  Invalid launch or termination notification function
 *          passed.
 *       0      Success.  Launch/Termination notifiers were installed.
 ************************************************/

int InstallLaunchTerminationNotifierFunctions(
        const LaunchNotificationCallBack YourLaunchNotificationFunction,
        const TerminationNotificationCallBack
                    YourTerminationNotificationFunction,
        OSStatus* ErrorReturnValue,
        EventHandlerRef* EventHandlerRemovalHandle)
{
    EventHandlerUPP launchTerminateCallbackUPP;
    EventTypeSpec eventsToListenFor [2];
    OSStatus functionResult;
    struct LaunchTerminationCallbackInfoStructure*
                    LaunchTerminationCallBackFunctionInfo;

    // --- Checking input arguments to ensure they are valid --- //
    //Checking the launched and termination callback functions passed to
    //make sure they are valid (i.e. not null).
    if ((YourLaunchNotificationFunction == NULL) ||
        (YourTerminationNotificationFunction == NULL))
    {
        return(-1); //Invalid arguments passed.  Returning error
    }

    //Setting output value to known value.
    *ErrorReturnValue = 0;

    // --- Getting context information for proxyCallbackFunction --- //

    /* Before we install the event handler we will create
     * the context information to pass to the proxy callback function.
     * The context information will be passed to the
     * LaunchTerminationProxyCallbackFunction.  The context information
     * holds the addresses of the actual functions the proxycallback function
     * should call on launch or termination of an application.
     * Note that we can't deallocate until we remove the handler.  This
     * is because the info is passed to the proxy callback function.
     */
     LaunchTerminationCallBackFunctionInfo =
        (struct LaunchTerminationCallbackInfoStructure*)
        malloc(sizeof(struct LaunchTerminationCallbackInfoStructure));

     if (LaunchTerminationCallBackFunctionInfo == NULL)
     {
        //if were unable to allocate structure to hold launch/termination
        //callback information then give up here.
        return(-2);
     }

    /* Now adding the callback information passed to this function to the
     * callback structure. The callback structure will be passed to the
     * LaunchTerminationProxyCallbackFunction.
     */
    LaunchTerminationCallBackFunctionInfo->launchCallback
                    = YourLaunchNotificationFunction;

    LaunchTerminationCallBackFunctionInfo->terminationCallback
                    = YourTerminationNotificationFunction;
```

```
// --- Installing the callbacks --- //

/* The first step before installing any callbacks is
 * to get the UPP of the function which will eventually be called.
 * The UPP is just a pointer to a function which we can use in the
 * functions.  We get the necessary UPP using the call NewEventHandlerUPP.
 * First Argument: The name of the function which will be called.  In this
 * case we want to use our proxy callback function.  The proxy callback
 * function will in turn call our Launch Termination notifier functions.
 * Return Value: A UPP which is just a pointer to the function passed in.
 * In this case a pointer to our proxy callback function.
 */
launchTerminateCallbackUPP =
    NewEventHandlerUPP(LaunchTerminationNotifierProxyCallbackFunction);

if (launchTerminateCallbackUPP == NULL)
{
    //ensure UPP is non-null.  If its null fail here.
    return(-3);
}

/* When installing the event handler first we need to define the class
 * and type of events we are listening for.  Launch notification events
 * come through ApplicationClass events and the AppLaunched kind.
 * Termination notification events come through ApplicationClass events and
 * the AppTerminated kind.  We must put this information into an array of
 * type EventTypeSpec.  We do this below.  The resulting array is used
 * in the InstallApplicationEventHandler call
 * below.
 */

//First event we are listening for is the launched event.
//Add this to the array of events to list of events to
//listen for.

    // event class to listen to is ApplicationClass
eventsToListenFor[0].eventClass = kEventClassApplication;
    // event kind to listen to is AppLaunched
eventsToListenFor[0].eventKind = kEventAppLaunched;

//Second event we are listening for is the termination event.
//Also add this to the array of events.

    // event class to listen to is ApplicationClass
eventsToListenFor[1].eventClass = kEventClassApplication;

    // event kind to listen to is AppTerminated
eventsToListenFor[1].eventKind = kEventAppTerminated;

/* Now installing the event handler.  What this causes is when the
 * application run loop is running any time an application is
 * launched/terminated the   specified callback gets called.
 * In this case the callback is the
 * LaunchTerminationNotifierProxyCallbackFunction function
 * defined above.  When called the proxy callback function in
 * turn will call the specified launch/termination handler function
 * which was passed as an argument to this function.  We install the
 * event handler with a call to InstallApplicationEventHandler.
 * First Argument: A UPP which points to the callback function we want to
 *     be called.  In this case we use the UPP created earlier which
 *     points to the proxy callback function.
 * Second Argument: The number of events we are currently listening for.
 *     We will be listening for two types: Launch events and
 *     Termination events.  Thus the number of events we are listening
```

```
     *     for is two.
     * Third Argument: A pointer to the list of events to listen for.  In
     *     this case we pass the array we allocated and filled out earlier.
     *     The array has two events described in it the kEventAppLaunched
     *     event and the kEventAppTerminated event.
     * Forth Argument: An parameter which will be passed to the event
     *     handler function.  In this case the handler is the proxy
     *     callback function defined above.  Here we pass a structure which
     *     contains the function pointers to our launch/termination callback
     *     functions.
     * Fifth Argument: On return this parameter would hold a reference
     *     which can be used to remove the callback function.  Since we
     *     need to remove the callback in the remove routine (below)
     *     we store this in a global variable for use later.
     * Return Value: An system error return value indicating result of the
     *     function.  The value will be zero (noErr) on success.
     */
    functionResult = InstallApplicationEventHandler(launchTerminateCallbackUPP,
                                    2,eventsToListenFor,
                                    LaunchTerminationCallBackFunctionInfo,
                                    EventHandlerRemovalHandle);

    //Done with UPP so dispose of it.
    DisposeEventHandlerUPP(launchTerminateCallbackUPP);

    if (functionResult != noErr)
    {
        //error installing event handler.
        //fail here and return error.
        *ErrorReturnValue = functionResult;
        return(-4);
    }

    return(0); //if got this far were successful.
}


/*************************************************
 * RemoveLaunchTerminationNotifierFunctions
 *************************************************
 * Purpose:  This function will remove the launch/terminate
 * notification functions.  On successful removal the launch/terminate
 * notification callback functions will no longer be called.
 *
 * Parameters:
 *  EventHandlerRemovalHandle   A pointer to a EventHandlerRef.
 * When calling RemoveLaunchTerminationNotifierFunctions this variable
 * should point to a EventHandlerRef which was returned from
 * InstallLaunchTerminationNotifierFunctions.  After calling this function
 * on success the event handler will be invalid.  Also, the callback functions
 * associated with the EventHandlerRef will be removed.
 *
 *  <Function Result>     A system error result as defined in MacErrors.h
 *
 *************************************************/
OSStatus RemoveLaunchTerminationNotifierFunctions(
                                EventHandlerRef* EventHandlerRemovalHandle)
{
    return(RemoveEventHandler(*EventHandlerRemovalHandle));
}


/*************************************************
 * main
```

```
 ************************************************
 * Purpose:  Demonstration of using Launch Termination notifiers.
 ************************************************/
int main (void)
{
    OSStatus Error;
    int Result;
    EventHandlerRef CallBackRemovalHandle;

    /* Installing the launch/termination notifiers into the current
     * Application event loop.
     * The notifier functions will be called whenever an application
     * is launched or terminated.
     * First Argument: The launch notification function which will be
     *    called on application launch.  Replace this with your own
     *    function if you like.
     * Second Argument: The termination notification function which will
     *    be called on application termination.  Replace this with your
     *    own function if you like.
     * Third Argument: On calling InstallLaunchTerminationNotifierFunctions
     *    this function must be a pointer to a preallocated OSStatus variable.
     *    On return this will have a error result if there was any in the
     *    functions.
     * Return Value: Integer representing if function was successful
     *    or not.  Value is zero on successful.
     */
    Result = InstallLaunchTerminationNotifierFunctions(LaunchFunction,
                    TerminateFunction, &Error, &CallBackRemovalHandle);

    if (Result != 0) //if couldn't install notifier then give up.
    {
        return(1);
    }

    RunApplicationEventLoop ();//Process carbon events until time to quit

    RemoveLaunchTerminationNotifierFunctions(&CallBackRemovalHandle);

    return(0);
}/*end main */
```

Listing 1. Registering for launch/termination notifications using Carbon Events

Note that all the above discussion is only valid in Mac OS9 CarbonLib 1.3.1 and later or Mac OS X 10.0 and later.

Back to top

Cocoa NSWorkspace Notifications

Another method of determining process lifetime without polling is to use Cocoa's NSWorkspace notifications. Specifically, by registering for `NSWorkspaceDidLaunchApplicationNotification` and `NSWorkspaceDidTerminateApplicationNotification` you will be notified when any new processes is launched or terminated. This approach is demonstrated in the code listing below. Note that the code works in Mac OS X (10.0 and later).

The method you will be interested in code listing is the `init` method. This method takes selectors which represent the methods which will be called when a notification arrives. The code uses generic methods `LaunchNotificationMethod` and `TerminateNotificationMethod` to receive the notifications. You will likely want to change or replace these generic methods with your own. The `init` call will add the notifications to the current application main event loop. Note that before the notifications are registered the application must have an active connection to the window server. If your application doesn't yet have a connection one can be created by calling `[NSApplication sharedApplication]`. Note that the ApplicationEventLoop must be run in order for the notification methods to be called properly.

Note:
Unlike the Carbon notifications (discussed above) the Cocoa and Java NSWorkspace notifications do not notify on all processes which have a window server connection. The NSWorkspace notifications do not fire for applications which have certain keys set in their Info.plist description file. Some examples of the keys that prevent NSWorkspace notifications are: `LSBackgroundOnly`, `LSUIElement`. This property of the NSWorkspace notifications is subject to change in the future.

Note as mentioned earlier the Cocoa notifications will only fire for processes which require a connection to the window server. That is processes which use the higher-level frameworks in some way. This means that certain types of processes (for example, low-level UNIX processes) won't cause Cocoa launch/termination notifications.

```
#import <AppKit/AppKit.h>

@interface LaunchTerminateNotifier : NSObject
{
}

-(void) LaunchNotificationMethod:
            (NSNotification *)LaunchNotificationRecieved;

-(void) TerminateNotificationMethod:
            (NSNotification *)TerminationNotificationRecieved;

-(id)init: (SEL) YourLaunchNotificationMethod:
           (SEL) YourTerminationNotificationMethod;

@end

@implementation LaunchTerminateNotifier

/*****************************************************
 * init
 *****************************************************
 * Purpose:  This method will setup the LaunchTerminateNotifer to notify
 * the specified methods.  After this method is called and the application
 * loop run you will see notifications on launch and termination of
 * applications.  Note that the notifications only apply to 'regular'
 * processes that launch and terminate.  That is, processes which use
 * the window server.  Thus, these notifications do not apply to
 * low-level UNIX type processes.
 *
 * Parameters:
 *  YourLaunchNotificationMethod        A selector which
 * represents a method with one argument:  a pointer to a NSNotification.
 * Before calling init the YourLaunchNotificationMethod parameter is
 * expected to have a selector pointing to a method which the user desires
 * to be called any time a process launches.  This is the method which will
 * be called when any recognized application launches.
 *
 *  YourTerminationNotificationMethod   A selector which
 * represents a method with one argument:  a pointer to a NSNotification.
 * Before calling init the YourTerminationNotificationMethod parameter is
 * expected to have a selector pointing to a method which the user desires
 * to be called any time a process launches.  This is the method which will
 * be called when any recognized application terminates.
 *
 * Additional Note: For these notifications to work you must have an active
 * connection with the window server.  This can be done by calling
 * [NSApplication sharedApplication] *before* registering for notifications.
 * Also, here we must run the resulting Application event loop to ensure
 *  we get the notifications properly.
 *
 *****************************************************/
```

```
-(id)init: (SEL) YourLaunchNotificationMethod:
           (SEL) YourTerminationNotificationMethod
{
    NSWorkspace* CurrentAppsSharedWorkspace;
    NSNotificationCenter* WorkspaceNotificationCenter;

    //Calling init on super as always
    self = [super init];

    /* Before setting up the notifications we first must get access to the
     * workspace notification center for this application.
     * This is because all NSWorkspace notifications are posted to
     * NSWorkspace's own notification center, not the application's default
     * notification center.
     */

    /* Method call: To get access to the notification center for our the
     * workspace we must first get access to the shared workspace.  We
     * call the method sharedWorkspace on NSWorkspace to do this.  This
     * call will return the shared workspace for our application.
     */

    CurrentAppsSharedWorkspace = [NSWorkspace sharedWorkspace];

    /* Method Call: Once we have the workspace for our application we can ask
     * for the workspace's notification center.  We do this by calling the
     * noticationCenter method on the shared workspace for our application.
     * This will return the notification center for our the workspace.
     */

    WorkspaceNotificationCenter =
               [CurrentAppsSharedWorkspace notificationCenter];

    /* Here we are registering for the launch notifications.  This is done
     * by registering for the NSWorkspaceDidLaunchApplicationNotification
     * inside our the NSWorkspace notification center.  We do the registration
     * using the method call addObserver on the workspace's notification center.
     * First Argument: The observor who will receive the notifications.  In
     *     this case we want us to receive the notifications so add 'self'
     *     as the observor.
     * Second Argument: The selector for the method which will be called when
     *     a notification arrives.  In this case it is the selector passed into
     *     this method. Note the method being called must have one argument
     *     which is a NSNotification.
     * Third Argument: The name notification we are registering for.
     *     In this case we want launch notifications.
     *     The name of the notification for launch notifications is represented
     *     by the NSWorkspace constant:
     *     NSWorkspaceDidLaunchApplicationNotification
     * Forth Argument: An object the notification must be associated with for
     *     us to be notified.  Here we don't need or want any extra selectors
     *     since we want all launch notifications.  Thus, we pass nil for this
     *     optional argument so that we receive all application
     *     launch notifications.
     */

    [WorkspaceNotificationCenter addObserver:self
    selector:YourLaunchNotificationMethod
    name:NSWorkspaceDidLaunchApplicationNotification object:nil];

    /* Here we are registering for the termination notifications.  This is done
     * by registering for the NSWorkspaceDidTerminateApplicationNotification
     * inside our applications notification center.  We do the registration
     * using the method call addObserver on the workspace's notification center
```

```
     * First Argument: The observor who will receive the notifications.  In
     *     this case we want us to receive the notifications so add 'self'
     *     as the observor.
     * Second Argument: The selector for the method which will be called when
     *     a notification arrives.  In this case it is the selector passed into
     *     this method. Note the method being called must have one argument
     *     which is a NSNotification.
     * Third Argument: The name notification we are registering for.
     *     In this case we want termination notifications.
     *     The name of the notification for terminate notifications is
     *     represented by the NSWorkspace constant:
     *     NSWorkspaceDidTerminateApplicationNotification
     * Forth Argument: An object the notification must be associated with for
     *     us to be notified.  Here we don't need or want any extra selectors
     *     since we want all terminate notifications.  Thus, we pass nil
     *     for this optional argument so that we receive all application
     *     termination notifications.
     */
    [WorkspaceNotificationCenter addObserver:self
    selector:YourTerminationNotificationMethod
    name:NSWorkspaceDidTerminateApplicationNotification object:nil];

    return self; //as always in init we need to return ourself
}

/*************************************************
 * dealloc
 *************************************************
 * Purpose:  This method deallocates the LaunchTerminateNotifier object.
 * Also here we will unregister for the notifications.
 *
 * Parameters: No Parameters
 *
 *************************************************/
- (void)dealloc
{
    // --- Removing the notifications --- //

    /* Before removing the notifications we first must get access to
     * the workspace notification center for this application.
     * This is because the NSWorkspace is who we have to unregister with
     * for the notifications
     */

    /* Method call: To get access to the notification center for our the
     * workspace we must first get access to the shared workspace.  We
     * call the method sharedWorkspace on NSWorkspace to do this.  This
     * call will return the shared workspace for our application.
     */

    NSWorkspace* CurrentAppsSharedWorkspace = [NSWorkspace sharedWorkspace];

    /* Method Call: Once we have the workspace for our application we can ask
     * for the workspace's notification center.  We do this by calling the
     * noticationCenter method on the shared workspace for our application.
     * This will return the notification center for the workspace.
     */

    NSNotificationCenter* WorkspaceNotificationCenter =
               [CurrentAppsSharedWorkspace notificationCenter];

    /* Here we are unregistering for the launch and terminate notifications.
     * This is done by removing ourself as an observor for all notifications.
     * Since we will only be registered for launch and terminate notifications
```

```
     * this effectively removes us in a single method call.
     * First Argument: The observor who will remove from the notification
     *     list.  In this case we need to remove ourselves from the notification
     *     list.  Thus unregister 'self' for all notifications.
     */

    [WorkspaceNotificationCenter removeObserver:self];

    [super dealloc]; //as always we need to call dealloc on super when we
                     //are done.

}

/***************************************************
 * LaunchNotificationMethod
 ***************************************************
 * Purpose:  This is the method which will become active any time
 * a launch occurs on the system.  This is because it
 * was the method specified in the selector when calling
 * init.
 *
 * Parameters:
 *
 *  LaunchNotificationRecieved      A Notification
 * sent which is associated with an application launching.  The
 * notification contains information on the process just launched.
 *
 ***************************************************/

-(void) LaunchNotificationMethod: (NSNotification *)LaunchNotificationRecieved
{
    /* Looking up process name based upon the dictionary of
     * process information provided.  Currently this is the
     * only information in the list.  However, more is
     * scheduled to be added in the future.
     */
    NSString *LaunchedApplicationProcessName =
        [[LaunchNotificationRecieved userInfo]
                        objectForKey: @"NSApplicationName"];

    NSLog(@"Application Launched: %@\n",
                        LaunchedApplicationProcessName);
}

/***************************************************
 * TerminateNotificationMethod
 ***************************************************
 * Purpose:  This is the method which will become active any time
 * a termination occurs on the system.  This is because it
 * was the method specified in the selector when calling
 * init.
 *
 * Parameters:
 *
 *  TerminationNotificationRecieved     A Notification
 * sent which is associated with an application terminating.  The
 * notification contains information on the process just terminated.
 *
 ***************************************************/
-(void) TerminateNotificationMethod:
            (NSNotification *)TerminationNotificationRecieved
{
    /* Looking up process name based upon the dictionary of
     * process information provided.  Currently this is the
```

```
     * only information in the list.  However, more is
     * scheduled to be added in the future.
     */
    NSString *TerminatedApplicationProcessName =
        [[TerminationNotificationRecieved userInfo]
                        objectForKey: @"NSApplicationName"];

    NSLog(@"Application Terminated: %@\n",
                    TerminatedApplicationProcessName);
}

@end

/*************************************************
 * main
 *************************************************
 * Purpose:  Demonstration of using Launch Termination notifiers.
 *************************************************/
int main (int argc, const char * argv[])
{
    //need a auto release pool for allocating objects.
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    /* For the notifications to work properly we need to have an active
     * connection to the window server.  This is done here by calling
     * sharedApplication on the NSApplication.  In a regular Cocoa application
     * (as opposed to a tool) this will likely be done for you.  However,
     * we call sharedApplication and run the resulting NSApplication to
     * ensure we get the notifications properly
     */
    NSApplication* CurrentApplication = [NSApplication sharedApplication];

    /* Registering for the notifications using the LaunchTerminateNotifier
     * method init.  The two parameters to init are selectors for the launch
     * and termination methods which will be called on notification.
     * First Argument: A selector representing the method which
     *    will be called each time an application is launched.
     * Second Argument: A selector representing the method which
     *    will be called each time an application is terminated.
     */
    [[LaunchTerminateNotifier alloc] init:
                @selector(LaunchNotificationMethod:):
                @selector(TerminateNotificationMethod:)];

    /* Running the current applications main event loop.  This will allow us
     * to receive the launch/terminate notifications
     */
    [CurrentApplication run];

    // No more need for autorelease pool so release
    [pool release];
    return 0;
}
```

Listing 2.Registering for launch/termination notifications using Cocoa's NSWorkspace notifications

Cocoa-Java NSWorkspace Notifications

Similar to Cocoa's NSWorkspace notifications is the notifications provided byt the Cocoa-Java version of NSWorkspace. Again, by registering for `NSWorkspaceDidLaunchApplicationNotification` and `NSWorkspaceDidTerminateApplicationNotification` you will be notified when any new processes is launched or terminated. This approach is demonstrated in the code listing below. Note that the code works in Mac OS X (10.0 and

later).

The method you will be interested in code listing is the `RegisterForLaunchTerminateNotifications` method. This method takes NSSelectors which represent the methods which will be called when a notification arrives. The code uses generic methods `LaunchNotificationMethod` and `TerminateNotificationMethod` to receive the notifications. You will likely want to change or replace these generic methods with your own.

> **Note:**
> Unlike the Carbon notifications (discussed above) the Cocoa and Java NSWorkspace notifications do not notify on all processes which have a window server connection. The NSWorkspace notifications do not fire for applications which have certain keys set in their Info.plist description file. Some examples of the keys that prevent NSWorkspace notifications are: `LSBackgroundOnly`, `LSUIElement`. This property of the NSWorkspace notifications is subject to change in the future.

Note as mentioned earlier the Cocoa-Java notifications will only fire for processes which require a connection to the window server. That is processes which use the higher-level frameworks in some way. This means that certain types of processes (for example, low-level UNIX processes) won't cause Cocoa-Java launch/termination notifications.

```
import com.apple.cocoa.foundation.*;
import com.apple.cocoa.application.*;

public class LaunchTerminateNotify extends NSObject
{

/**************************************************
 * RegisterForLaunchTerminateNotifications
 **************************************************
 * Purpose:  This method will setup the LaunchTerminateNotifer to notify
 * the specified methods.  After this method is will see notifications
 * on launch and termination of applications.  Note that the notifications
 * only apply to 'application' processes that launch and terminate.
 * That is, processes which use the window server.  Thus, these
 * notifications do not apply to low-level UNIX type processes.
 *
 * Parameters:
 *  YourLaunchNotificationMethod        A NSSelector which
 * represents a method with one argument:  a NSNotification.
 * Before calling RegisterForLaunchTerminateNotifications the
 * YourLaunchNotificationMethod parameter is expected to have a selector
 * pointing to a method which the user desires to be called any time a
 * process launches.  This is the method which will be called when any
 * recognized application launches.
 *
 *  YourTerminationNotificationMethod   A NSSelector which
 * represents a method with one argument:  a NSNotification.
 * Before calling RegisterForLaunchTerminateNotifications the
 * YourTerminationNotificationMethod parameter is expected to have a selector
 * pointing to a method which the user desires to be called any time a
 * process terminates.  This is the method which will be called when any
 * recognized application terminates.
 *
 **************************************************/

public void RegisterForLaunchTerminateNotifications(
                    NSSelector YourLaunchNotificationMethod,
                    NSSelector YourTerminationNotificationMethod)
{
    NSWorkspace currentAppsSharedWorkspace;
    NSNotificationCenter workspaceNotificationCenter;

    /* Before setting up the notifications we first must get access to the
     * workspace notification center for this application.
     * This is because all NSWorkspace notifications are posted to
     * NSWorkspace's own notification center, not the application's default
     * notification center.
```

```
 */

/* Method call: To get access to the notification center for our the
 * workspace we must first get access to the shared workspace.  We
 * call the method sharedWorkspace on a new NSWorkspace instance to do this.
 * This call will return the shared workspace for our application.
 */

currentAppsSharedWorkspace = new NSWorkspace().sharedWorkspace();

/* Method Call: Once we have the workspace for our application we can ask
 * for the workspace's notification center.  We do this by calling the
 * noticationCenter method on the shared workspace for our application.
 * This will return the notification center for our the workspace.
 */

workspaceNotificationCenter =
            currentAppsSharedWorkspace.notificationCenter();

/* Here we are registering for the launch notifications.  This is done
 * by registering for the NSWorkspaceDidLaunchApplicationNotification
 * inside our the NSWorkspace notification center.  We do the registration
 * using the method call addObserver on the workspace's notification center.
 * First Argument: The observer who will receive the notifications.  In
 *     this case we want us to receive the notifications so add 'this'
 *     (i.e. ourselves) as the observer.
 * Second Argument: The NSSelector for the method which will be called when
 *     a notification arrives.  In this case it is the selector passed into
 *     this method. Note the method being called must have one argument
 *     which is a NSNotification.
 * Third Argument: The name notification we are registering for.
 *     In this case we want launch notifications.
 *     The name of the notification for launch notifications is represented
 *     by the String constant: "NSWorkspaceDidLaunchApplicationNotification"
 * Forth Argument: An object the notification must be associated with for
 *     us to be notified.  Here we don't need or want any extra selectors
 *     since we want all launch notifications.  Thus, we pass null for this
 *     optional argument so that we receive all application launch
 *     notifications.
 */

workspaceNotificationCenter.addObserver(this,
        YourLaunchNotificationMethod,
        "NSWorkspaceDidLaunchApplicationNotification", null);

/* Here we are registering for the termination notifications.  This is done
 * by registering for the NSWorkspaceDidTerminateApplicationNotification
 * inside our applications notification center.  We do the registration
 * using the method call addObserver on the workspace's notification center
 * First Argument: The observer who will receive the notifications.  In
 *     this case we want us to receive the notifications so add 'this'
 *     (i.e. ourselves) as the observer.
 * Second Argument: The NSSelector for the method which will be called when
 *     a notification arrives.  In this case it is the selector passed into
 *     this method. Note the method being called must have one argument
 *     which is a NSNotification.
 * Third Argument: The name notification we are registering for.
 *     In this case we want terminate notifications.
 *     The name of the notification for terminate notifications is
 *     represented by the String constant:
 *                  "NSWorkspaceDidTerminateApplicationNotification"
 * Forth Argument: An object the notification must be associated with for
 *     us to be notified.  Here we don't need or want any extra selectors
 *     since we want all terminate notifications.  Thus, we pass null for
```

```
     *     this optional argument so that we receive all application
     *     terminate notifications.
     */

    workspaceNotificationCenter.addObserver(this,
            YourTerminationNotificationMethod,
            "NSWorkspaceDidTerminateApplicationNotification", null);

}

/**************************************************
 * UnRegisterForLaunchTerminateNotifications
 **************************************************
 * Purpose:  This method unregisters for the launch terminate
 * notifications.
 *
 * Parameters: No Parameters
 *
 **************************************************/
public void UnRegisterForNotifications()
{
    // --- Removing the notifications --- //

    /* Before removing the notifications we first must get access to
     * the workspace notification center for this application.
     * This is because the NSWorkspace is who we have to unregister with
     * for the notifications
     */

    /* Method call: To get access to the notification center for our the
     * workspace we must first get access to the shared workspace.  We
     * call the method sharedWorkspace on a new NSWorkspace instance to do this.
     * This call will return the shared workspace for our application.
     */

    NSWorkspace currentAppsSharedWorkspace = new NSWorkspace().sharedWorkspace();

    /* Method Call: Once we have the workspace for our application we can ask
     * for the workspace's notification center.  We do this by calling the
     * noticationCenter method on the shared workspace for our application.
     * This will return the notification center for the workspace.
     */

    NSNotificationCenter workspaceNotificationCenter =
               currentAppsSharedWorkspace.notificationCenter();

    /* Here we are unregistering for the launch and terminate notifications.
     * This is done by removing ourself as an observer for all notifications.
     * Since we will only be registered for launch and terminate notifications
     * this effectively removes us in a single method call.
     * First Argument: The observer who will remove from the notification
     *     list.  In this case we need to remove ourselves from the notification
     *     list.  Thus unregister 'this' for all notifications.
     */

    workspaceNotificationCenter.removeObserver(this);
}

/**************************************************
 * launchNotificationMethod
 **************************************************
 * Purpose:  Once the notifications are installed this function
 * will be called any time an application launch occurs on the
 * system.  This is because it was the method specified in the
```

```
 * NSSelector when calling RegisterForLaunchTerminateNotifications
 * Note that since we are talking about processes we
 * mean "normal" processes.  These are processes which link with
 * or use Carbon or Cocoa in some way.  This excludes processes
 * which are launched which low-level UNIX type processes.
 *
 * Parameters:
 *  LaunchNotificationreceived      A Notification
 * sent to this routine which describes the launch.  The
 * notification contains information on the process just launched.
 *
 ************************************************/

public void launchNotificationMethod(NSNotification LaunchNotificationreceived)
{
    /* Looking up process name based upon the dictionary of
     * process information provided.  Currently this is the
     * only information in the list.  However, more is
     * scheduled to be added in the future.
     * We pull the information form the diction using the objectForKey
     * on the dictionary returned from userInfo()
     */
    String launchedApplicationProcessName = (String)
        LaunchNotificationreceived.userInfo().objectForKey("NSApplicationName");

    //Printing out the application name of the application that launched.
    System.out.println("Application Launched: " + launchedApplicationProcessName);
}

/************************************************
 * terminateNotificationMethod
 ************************************************
 * Purpose:  Once the notifications are installed this function
 * will be called any time an application termination occurs on the
 * system.  This is because it was the method specified in the
 * NSSelector when calling RegisterForLaunchTerminateNotifications
 * Note that since we are talking about processes we
 * mean "normal" processes.  These are processes which link with
 * or use Carbon or Cocoa in some way.  This excludes processes
 * which are terminated which low-level UNIX type processes.
 *
 * Parameters:
 *  TerminationNotificationreceived     A Notification
 * sent to this routine which describes the termination.  The
 * notification contains information on the process just terminated.
 *
 ************************************************/
public void terminateNotificationMethod(
                NSNotification TerminationNotificationreceived)
{
    /* Looking up process name based upon the dictionary of
     * process information provided.  Currently this is the
     * only information in the list.  However, more is
     * scheduled to be added in the future.
     * We pull the information form the diction using the objectForKey
     * on the dictionary returned from userInfo()
     */
    String terminatedApplicationProcessName = (String)
   TerminationNotificationreceived.userInfo().objectForKey("NSApplicationName");

    //Printing out the application name of the application that launched.
    System.out.println("Application Terminated: "
                        + terminatedApplicationProcessName);
}
```

```
/**************************************************
 * awakeFromNib
 **************************************************
 * Purpose:  This routine gets called when the application
 * is initalized.  This allows us to register for the
 * notifications.
 *
 * Parameters: None
 *
 **************************************************/
public void awakeFromNib()
{
    NSSelector launchMethodSelector;
    NSSelector terminationMethodSelector;

    System.out.println("Now launch and terminate applications\n");

    /* Registering for the notifications using the LaunchTerminateNotifier
     * method RegisterForLaunchTerminateNotifications.  The two parameters to
     * init are NSSelectors for the launch and termination methods which will
     * be called on notification.  Here we create the NSSelectors using the
     * NSSelector constructor routine.
     * First Argument: The name of the function which will be called expressed
     *    as a String.  Here the name of the method we want called when
     *    launching occurs.
     * Second Argument: The type of the parameters passed to the function.
     *    Here there is only one parameter passed to these routines a
     *    single NSNotification.
     */
    launchMethodSelector =
            new NSSelector("launchNotificationMethod",
                           new Class[] {NSNotification.class});

    /* Here we need another NSSelector for the termination method which gets
     * called any time a application termination is detected.
     * First Argument: The name of the function which will be called expressed
     *    as a String.  Here the name of the method we want called when
     *    termination occurs.
     * Second Argument: The type of the parameters passed to the function.
     *    Here there is only one parameter passed to these routines a
     *    single NSNotification.
     */
    terminationMethodSelector =
            new NSSelector("terminateNotificationMethod", n
                           ew Class[] {NSNotification.class});

    /* Now that we have the NSSelectors for the routines we want called
     * we can register for the notifications using the
     * RegisterForLaunchTerminateNotifications routine.
     * First Argument: A selector representing the method which
     *    will be called each time an application is launched.
     * Second Argument: A selector representing the method which
     *    will be called each time an application is terminated.
     */
    RegisterForLaunchTerminateNotifications(
                launchMethodSelector, terminationMethodSelector);
}

}//end class
```

Listing 2. Registering for launch/termination notifications using Cocoa-Java's NSWorkspace notifications

Note that all the above discussion relating to NSWorkspace notifications is valid in Mac OS X 10.0 and later.

Back to top

Mach Port Dead-Name Notifications

Another method of determining process lifetime without polling is to use Mach port dead-name notifications. In this method you register on a specific process using its Mach port. When that process terminates the Mach port associated with that process is automatically deallocated and invalidated by the Mac OS X kernel. When invalidation occurs the Mac OS X kernel will also notify any interested parties that the port was invalidated.

You can register for these notifications using the Core Foundation call `CFMachPortSetInvalidationCallBack()` which registers a callback function with Core Foundation. Note this method will only provide termination notifications and not launch notifications with the process you register upon. The Mach port notification approach is demonstrated in the code listing below. Note that the code works in Mac OS X (10.0.4 and later).

The function you will be interested in code listing is the `InstallMachTerminationNotifier()` function. This function takes a function pointer to the termination callback function as well as a PID which represents the process you want to notify against. The specified termination callback function will be called when the process with the given PID terminates. The code uses the generic function `TerminateFunction()` to receive the callbacks. However, you will likely want to change or replace this function. The `InstallMachTerminationNotifier()` will add the notification callbacks to the run loop source returned from this function. The resulting run loop source must be added to a run loop and then the resulting run loop run for the callbacks to be called properly. Note the `InstallMachTerminationNotifier()` function is re-enterant and can be called multiple times for multiple different processes you wish to register upon.

> **Note:**
> This method of process notification does have a security restriction to consider. The Mach port notification method requires a lookup of the process task of the process you are interested in. This lookup can only be performed if your process has a UID matching the process you are attempting to be notified against. This means that you can only notify on other processes which are owned by the same user as your process. Alternately, if your process is running as root it can get notification on any running process (regardless of the process UID).

Unlike the previous solutions in Carbon,Java and Cocoa the Mach port method can register for notifications on any process on the system (including low-level UNIX processes) assuming you have the correct permissions to do so.

```
#include <stdio.h>
#include <mach/mach.h> //including necessary Mach port definitions
#include <CoreFoundation/CoreFoundation.h>

typedef void (*TerminationCallBack) (pid_t PIDOfTerminatedProcess);

struct TerminationExtraInfoStructure
{
    pid_t       ProcessWhichTerminated;
    TerminationCallBack TerminationFunctionToCall;
};

/*************************************************
 * TerminationProxyCallbackFunction
 *************************************************
 * Purpose:  This callback function will be called when a process
 * we are watching terminates.  Note this only occurs after
 * InstallMachTerminationNotifier has been called and the returned
 * CFRunloopsource is executed.  This function will call the
 * specified termination callback function which was specified
 * in the InstallMachTerminationNotifier call.
 *
 * Parameters: (All parameters are required if this function is to
 * be called)
 *
 *  port        The CFMachPort which is associated with the
 * Mach port which caused the notification.  This is also the
 * CFMachPort which the run loop source was created from.
 *
```

```
 *  info        A generic pointer pointing to data associated
 * with the callback.  This is the extra context information we
 * created in the InstallMachTerminationNotifier call.  It is used
 * to find the callback function we need to call as well as return
 * the process which terminated PID.
 ************************************************/
void TerminationProxyCallbackFunction(CFMachPortRef port, void *info)
{
    #pragma unused (port)

    struct TerminationExtraInfoStructure* extraTerminationInfo;
    TerminationCallBack terminationFunctionToCall;
    pid_t pIDOfProcessWhichCausedNotification;

    /* Getting extra termination information from the info
     * passed to this function.  This contains both the
     * function to call the process PID which caused
     * the notification.
     */
    extraTerminationInfo =
            (struct TerminationExtraInfoStructure*)info;

    //Getting the termination notification callback from
    //the extra info structure passed to this function.
    terminationFunctionToCall =
                extraTerminationInfo->TerminationFunctionToCall;

    if (terminationFunctionToCall == NULL)
    {
        //if the termination notification functions gathered
        //is invalid then we need to fail here.
        return;
    }

    //Finding out the process PID of the terminated process
    //from the extra information structure.
    pIDOfProcessWhichCausedNotification =
            extraTerminationInfo->ProcessWhichTerminated;

    terminationFunctionToCall(pIDOfProcessWhichCausedNotification);
}

/************************************************
 * InstallMachTerminationNotifier
 ************************************************
 * Purpose:  This function will install the given termination
 * notification function.  On successful install the termination
 * notification function will be called when the given process
 * terminates.
 *
 * Parameters:
 *  PIDOfProcessWeAreInterestedIn       A process pid.
 * Before calling InstallMachTerminationNotifier this parameter is expected
 * to have a PID of the process you wish to notify against.
 *
 *  YourTerminationCallbackFunction     A function pointer to a
 *
 *  RunloopSourceReturned           When calling
 * InstallMachTerminationNotifier this variable should be a pre-allocated
 * CFRunLoopSourceRef.  On return from InstallMachTerminationNotifier
 * this variable will hold a RunLoopSource which must be added to
 * a run-loop source and then run for the termination callback to be called.
 *
 *  KernError           A kern_return result.
```

```
 * InstallMachTerminationNotifier this variable should be a pre-allocated
 * kern_return_t.  On return from InstallMachTerminationNotifier
 * this variable will hold any kernel result codes caused by any
 * errors the function encountered.  These result codes are defined
 * in /usr/include/mach/kern_return.h.  Note that if the function
 * result is zero (success) then KernError will also be zero
 * (KERN_SUCCESS)
 *
 *      A integer return value.
 *              See result codes listed below.
 *      Result Codes:
 *      -8  Error creating runloop source based on
 *          CFMachPort.
 *      -7  Error deallocating our copy of the Mach port
 *          for the process.
 *      -6  Error creating CFMachPort based on processes
 *          Mach port.
 *      -5  Error Allocating extra context info for termination
 *                      information.
 *      -4  Unable to get task for process with given PID.
 *          Note this common error is generally caused by the fact
 *                      you need to be root or have the same UID as
 *          the process you are trying to lookup.  Another
 *                      alternative is if the PID you passed in is for
 *                      a process which doesn't exist.
 *      -3  Error getting task for our own process.
 *      -2  Invalid callback function passed.
 *      -1  Invalid PID of process to register for callbacks on.
 *       0      Success.  Termination callback were installed.
 *
 * Additional Note: Yes, this function is re-enterant.  You can call it
 * each time and get different runloop sources each time.
 *
 ************************************************/

int InstallMachTerminationNotifier(
        const pid_t PIDOfProcessWeAreInterestedIn,
        const TerminationCallBack YourTerminationCallbackFunction,
        CFRunLoopSourceRef* RunloopSourceReturned,
        kern_return_t* KernError)
{
    kern_return_t   kernResult;
    mach_port_t     taskOfOurProcess;
    mach_port_t     machPortForProcess;
    Boolean     ShouldFreePort;
    CFMachPortRef   CFMachPortForProcess;
    CFMachPortContext TerminationCallbackContextInfo;
    struct TerminationExtraInfoStructure*
                TerminationCallBackFunctionInfo;

    //Setting output to success value to start with.
    *KernError = 0;
    *RunloopSourceReturned = NULL;

    // --- Checking input arguments --- //

    //A PID less than one is an invalid PID
    if (PIDOfProcessWeAreInterestedIn < 1)
    {
        return(-1);
    }

    //Checking the callback function to ensure its valid
    //basically we ensure its non-null.
```

```
    if (YourTerminationCallbackFunction == NULL)
    {
        return(-2);
    }

    // --- Doing lookup of port associated with process --- //

    /* The first step is to get the task for our own process.  We do this
     * with the call mach_task_self() which returns the task associated with
     * our own process.  We need the task for our own process for future
     * lookups.  Note we don't need to deallocate the returned Mach port.
     */

    taskOfOurProcess = mach_task_self();

    if (taskOfOurProcess == MACH_PORT_NULL)
    {
        //error doing task lookup for ourself so give up.
        return(-3);
    }

    /* Next we need to do a lookup of the task we are interested in.
     * We need the task because it is required to lookup the port of the given
     * process.  We lookup the task of the process we are interested in using
     * task_for_pid call.
     * First Argument: Which task we wish to do the lookup from.  Of course we
     *     want to do the lookup from own process.  Thus, pass task for our
     *     own process.
     * Second Argument: The PID of the process we are interested in.  In this
     *    case we use the PID passed into this function call.
     * Third Argument: The actual task returned.  This is the task associated
     *    with the process we are interested in.  In this case the task
     *    is the Mach port for the process.
     * Return Value: The return value is a kernel result code defined by the
     *    system
     * Note: The returned task must be deallocated with mach_port_deallocate
     */

    kernResult = task_for_pid(taskOfOurProcess,
                    PIDOfProcessWeAreInterestedIn, &machPortForProcess);

    /* if were unsuccessful than return result and give up.  Note if you
     * are failing here it may be caused by the fact you need to be
     * root or have the same UID as the process you are trying to lookup.
     * Another alternative is if the Process PID you passed is for
     * a process which isn't running (doesn't exist).
     */
    if (kernResult != KERN_SUCCESS)
    {
        *KernError = kernResult; //setting output error
        return(-4);
    }

   // --- Creating Dynamic Store Context --- //

    /* Before we create the CFMachPort we will create the context
     * information.  The context information we will store
     * is the PID of the process which the Mach port is associated with.
     * First step is to allocate the extra info structure to put in the
     * CFMachPort.  This context information will be in the callback
     * function call to the proxy callback function.
     * Note that we can't deallocate this since it will be pointed to from
     * the MachPort and the Runloop source itself later.
     */
```

```
 TerminationCallBackFunctionInfo =
                malloc(sizeof(struct TerminationExtraInfoStructure));

 if (TerminationCallBackFunctionInfo == NULL)
 {
    //if were unable to allocate structure to hold termination callback
    //information then give up here.
    return(-5);
 }

/* Now adding the callback information passed to this function to the
 * "TerminationExtraInfoStructure".
 */
TerminationCallBackFunctionInfo->ProcessWhichTerminated
                    = PIDOfProcessWeAreInterestedIn;

TerminationCallBackFunctionInfo->TerminationFunctionToCall
                    = YourTerminationCallbackFunction;

/* Now adding the allocated structure to the dynamic store context.  This
 * context will be added to the dynamic store when it is created
 * Note that for all except the info field the context is null since we
 * don't care about any of these fields.
 */

TerminationCallbackContextInfo.version = NULL;
TerminationCallbackContextInfo.info =
                    (void*) TerminationCallBackFunctionInfo;
TerminationCallbackContextInfo.retain = NULL;
TerminationCallbackContextInfo.release = NULL;
TerminationCallbackContextInfo.copyDescription = NULL;

/* Now creating the CFMachport which will be returned.  We are creating the
 * CFMachPort based on an existing Mach port.  We create the CFMachPort
 * with the call CFMachPortCreateWithPort.
 * First Argument : The allocator to use in creating the CFMachPort.  As
 *    usual we want the default allocator so pass null.
 * Second Argument: The actual Mach port to use in creating the CFMachPort.
 *    In this case we are using the Mach port for the process we
 *    are interested in.
 * Third Argument: The callback to use when a message is received on the
 *    Mach port.  In this case we aren't interested in messages the Mach
 *    port is receiving.  Only if the port exists or not.  Thus, since
 *    we aren't interested in the messages we pass null for this
 *    parameter to not receive message callbacks.
 * Forth Argument: Context info to add the CFMachPort.  In this case we
 *    add the context information we created earlier.  This context
 *    information is returned back to the callback function which gets
 *    called.  In this case the proxy callback function.
 * Fifth Argument: On return this argument holds a boolean representing
 *    if we should free the Mach port.  We use this boolean for
 *    determination later if we should call mach_port_deallocate on
 *    the Mach port.
 * Return Value: A CFMachPort which was created using the given Mach
 *    port. Here it is the actual CFMachPort we want to return.
 *    On error the  result will be null however, this is the return
 *    value we want in case of error anyways.
 *
 */

CFMachPortForProcess = CFMachPortCreateWithPort
                (NULL, machPortForProcess, NULL,
                &TerminationCallbackContextInfo, &ShouldFreePort);
```

```
    if (CFMachPortForProcess == NULL)
    {
        //attempt to deallocate Mach port before giving up
        mach_port_deallocate(taskOfOurProcess, machPortForProcess);
        return(-6);
    }

    /* Done with Mach port of the process.  Thus we
     * deallocate the associated Mach port.  Deallocation is done
     * with the call mach_port_deallocate.  However we only do
     * deallocation if the CFMachPortCreateWithPort indicates it
     * should be done.
     * First Argument: The task to do the deallocation from in this
     *    case we want to do deallocation from our task.
     * Second Argument: The Mach port being deallocated.  In this
     *    case we deallocate the Mach port which is the Mach
     *    port of the process were interested in.  Note we
     *    are only deallocating our copy here not the original.
     */
    if (ShouldFreePort == TRUE)
    {
        kernResult = mach_port_deallocate(taskOfOurProcess,
                                          machPortForProcess);

        //if were unsuccessful than return result and give up.
        if (kernResult != KERN_SUCCESS)
        {
            *KernError = kernResult;
            return(-7);
        }
    }

    /* Now setting the actual invalidation callback on the CFMachPort.
     * Here we are actually registering for notification with the
     * CFMachPort.  The CFMachPort holds this information internally
     * and returns it as part of a runloop source it can generate.
     * We set the invalidation callback on the CFMachPort using the
     * call CFMachPortSetInvalidationCallBack.
     * First Argument: The actual CFMachPort we want to register the
     *    notification upon.  In this case the CFMachPort which is
     *    for the processes Mach port.
     * Second Argument: The calllback function we want to be called.
     *    In this case it is our proxy callback function defined above.
     */
    CFMachPortSetInvalidationCallBack
        (CFMachPortForProcess, TerminationProxyCallbackFunction);

    /* Now we will generate a runloop source based upon the CFMachPort we
     * currently have. This is done with the CFMachPortCreateRunLoopSource
     * call.  The runloop source returned will cause us to be notified when
     * the given process terminates (its Mach port becomes invalid)
     * First Argument: Allocator to use.   As usual we want default
     *     allocator so pass null.
     * Second Argument: The CFMachPort we wish to base the run loop
     *     source upon. Here we use the CFMachPort for the process we
     *     are interested in.
     * Third Argument: The 'order' of the CFRunloop source.  We want
     *     default behavior here too so pass zero.
     * Return value: the run loop source created.  Note that the
     *     return value must eventually be released.
     */
   *RunloopSourceReturned =  CFMachPortCreateRunLoopSource
                    (NULL, CFMachPortForProcess, (CFIndex) 0);
```

```
   //error creating runloop source so give up.
   if (*RunloopSourceReturned == NULL)
   {
        return(-8);
   }

    //Done with the CFMachPort so release it.
    CFRelease(CFMachPortForProcess);

    return(0); //if got this far were successful
}

/*****************************************************
 * TerminateFunction
 *****************************************************
 * Purpose:  This is the callback function which gets called any
 * time a specified process terminates.  Note before this
 * function will be called InstallMachTerminationNotifier must be called.
 * Also, the CFRunLoopSource returned from InstallMachTerminationNotifier
 * call must be executed.
 *
 * Parameters:
 *  PIDOfTerminatedProcess      A process pid.
 * this is the PID of the process which just terminated.
 *
 *****************************************************/
void TerminateFunction(pid_t PIDOfTerminatedProcess)
{
    //writing which process terminated.
    printf("Process Terminated, PID: %ld",
                (long) PIDOfTerminatedProcess);
    fflush(stdout); //making sure buffer is flushed.
}

/*****************************************************
 * main
 *****************************************************
 * Purpose:  Demonstrating the use of the Mach termination
 * notification.
 *
 * Important Note: You will need to edit the
 * kProcessWeAreNotifyingUpon constant to notify against
 * a process which you are following.
 *****************************************************/
int main (int argc, const char * argv[])
{
    CFRunLoopSourceRef RunloopSourceFromMachPort;
    kern_return_t Error;
    int Result;

    /* Here this is the process PID of the process we are notifying against
     * Note you need to know the processs PID you are targeting.  There
     * are several methods of determining process PID's.  One such method
     * is described in QA1123 which is listed at:
     * http://developer.apple.com/qa/qa2001/qa1123.html
     */
    const pid_t kProcessWeAreNotifyingUpon = 99;

    /* Installing the termination notifiers into the CFRunloop source.
     * The termination notifier function will be called whenever a
     * termination occurs on the specified process.
     * First Argument: The PID of the process we want to register for
     *    termination notifications upon.
     * Second Argument: The termination notification function which
```

```
     *    will be called.  Replace this with your own function if you like.
     * Third Argument: On return this will have a runloopsource which
     *    you can use to create a runloop which will call your notification
     *    function specified in the second argument.
     * Forth Argument: On return this will have a kernel return error which
     *    represents any kernel error encountered by the function.  Note
     *    if the return value from the call is success (zero) this
     *    will also be zero (Kern_success)
     * Return Value: Integer representing if function was successful
     *    or not.  Value is zero on success.
     */
    Result = InstallMachTerminationNotifier
        (kProcessWeAreNotifyingUpon, TerminateFunction,
        &RunloopSourceFromMachPort, &Error);

    if (Result != 0)
    {
        //Error installing callback so fail.
        printf("Error installing termination callback: %d\n", Result);
        return(1);
    }

    /* Adding a run loop source to the current run loop.
     * Here we are adding our termination notification to the current
     * run loop.
     * First Argument: The run loop to add the new source to.  In this
     *     case we want the default run loop for this process which is
     *     current run loop.
     * Second Argument: The run loop source which will be added to
     *     current run loop.  This is the run loop which we created earlier
     *     it contains termination notification information.
     * Third Argument: The run loop mode.  As usual we want default mode.
     *     Thus pass kCFRunLoopDefaultMode to get default mode.
     */
    CFRunLoopAddSource(CFRunLoopGetCurrent(),
                RunloopSourceFromMachPort, kCFRunLoopDefaultMode);

    //Done with run loop source so release.
    CFRelease(RunloopSourceFromMachPort);

    //Running the run loop.  This is required so that the
    //termination notification actually gets called.
    CFRunLoopRun();

    return 0;
}
```

Listing 3. Registering for termination notifications using Mach port dead-name notifications

Note that all the above discussion relating to Mach port dead-name notifications is valid in Mac OS X 10.0.4 and later.

Back to top

Summary

Process launch/terminate notifications are extreamly useful. Not only are they fairly easy to use but your users will love you for making use of them.

Back to top

References

Inside Mac OS X: Performance contains other hints on improving performance

QA 1123: Getting List of All Processes on Mac OS X contains information on getting a 'snapshot' of the process list

Back to top

Downloadables

	Acrobat version of this Note (120K)	Download
	Source project for Carbon Events solution (20K)	Download
	Source project for Cocoa NSWorkspace solution (16K)	Download
	Source project for Java NSWorkspace solution (20K)	Download
	Source project for Mach Port Invalidation solution (16K)	Download

Back to top
