The Birth of a Process Part-2

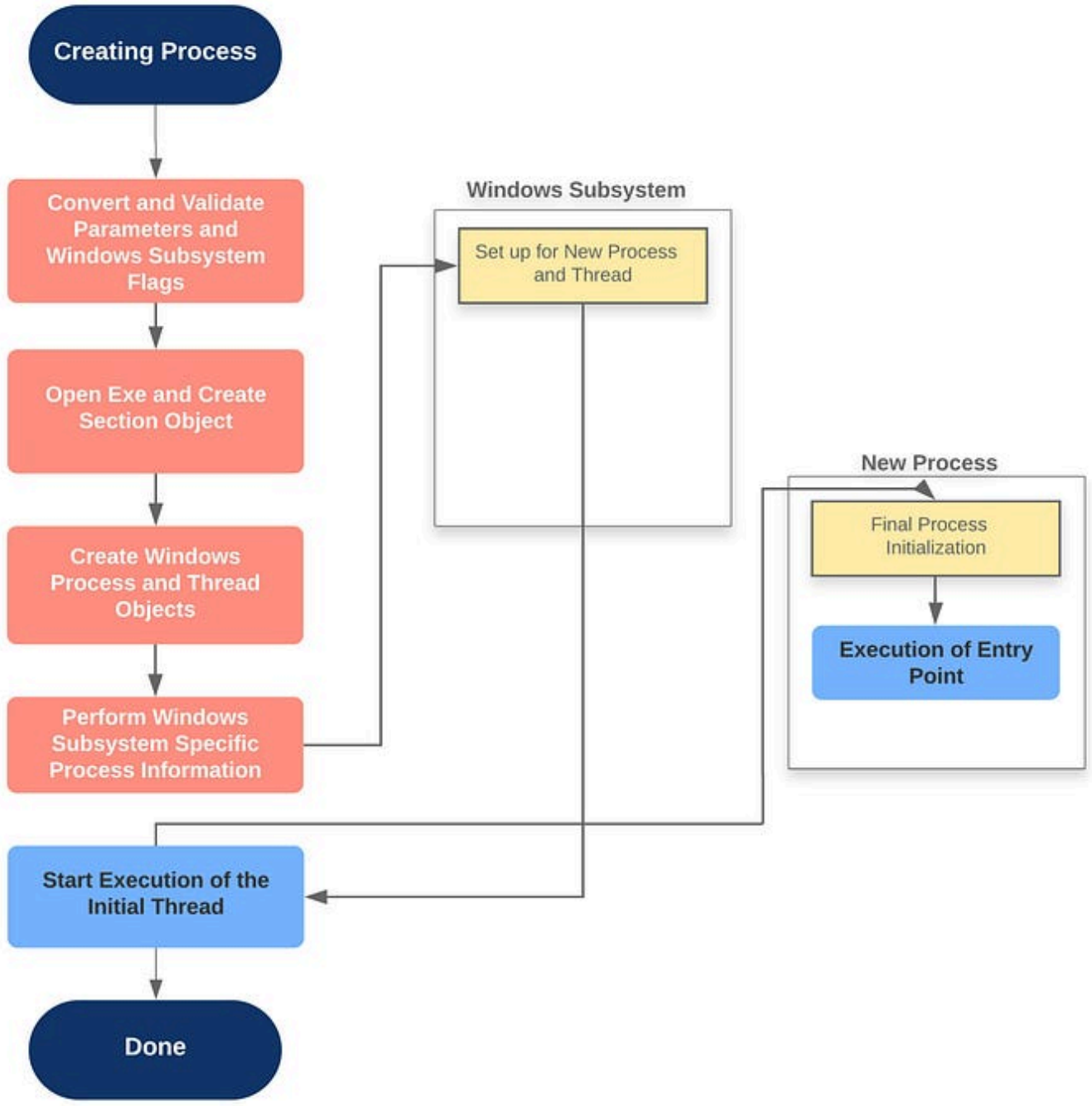
medium.com/@Achilles8284/the-birth-of-a-process-part-2-97c6fb9c42a2

Achilles April 30, 2020

Flow of CreateProcess

The creation of a Windows process (subsystem-specific) consists of several stages carried out in three sections of the Operating System: the Windows client-side library: Kernel32.dll, the Windows executive, and the Windows subsystem process.

The operations performed in each section is described through the diagram shown below:



Main Stages of Process Creation

Stage 1: Converting and Validating Parameter Flags

`CreateProcessInternalW` performs the following steps:

- The priority class for the new process is specified as independent bits in the CreationFlags parameter to the CreateProcess* functions.* Idle/Low (4)* Below Normal (6)* Normal (8)* Above Normal (10)* High (13)* Real-Time (24)Defaults to normal, if none specified.```Process creation won't fail if a Real-Time priority class is specified for the new process, whilst not having the Increase Scheduling Priority Privilege (SE_INC_BASE_PRIORITY_NAME), the High Priority class is used instead.`````` For a debug flag, Kernel32 will initiate a connection to the native debugging code in Ntdll.dll by calling `DbgUiConnectToDbg` and gets a handle to the debug object from the current TEB.```
- Supports multiple user-specified attributes. The attribute list passed on CreateProcess* calls permits passing back to the caller information beyond a simple status code, such as the TEB address of the initial thread, etc.(Important for protected processes | No query post creation)
- If the process is part of a job object, but the creation flags requested a separate virtual DOS machine (VDM), the flag is ignored.
- CreateProcessInternalW checks whether the process should be created as modern (with attributes: PROC_THREAD_ATTRIBUTE_ PACKAGE_FULL_NAME, PROC_THREAD_ATTRIBUTE_PARENT_PROCESS). If so, a call is made to the internal BaseAppXExtension to gather more contextual information on the modern app parameters described by a structure called APPX_PROCESS_CONTEXT.
- If the process is to be created as modern, the security capabilities (PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES) are recorded for the initial token creation by calling the internal BasepCreateLowBox function.
- If a modern process is created, then a flag is set to indicate to the kernel to skip embedded manifest detection. It's not needed in a modern process, it is already embedded.
- If the debug flag has been specified, then the `Debugger` value under the Image File Execution Options registry key for the executable is marked to be skipped.
- If no desktop is specified in the `STARTUPINFO` structure, the process is associated with the caller's current desktop.
- The application and command-line arguments passed to the API are analyzed and converted to the internal NT name if required.
- Most of the processed information is converted to a single large structure of type `RTL_USER_PROCESS_PARAMETERS` .

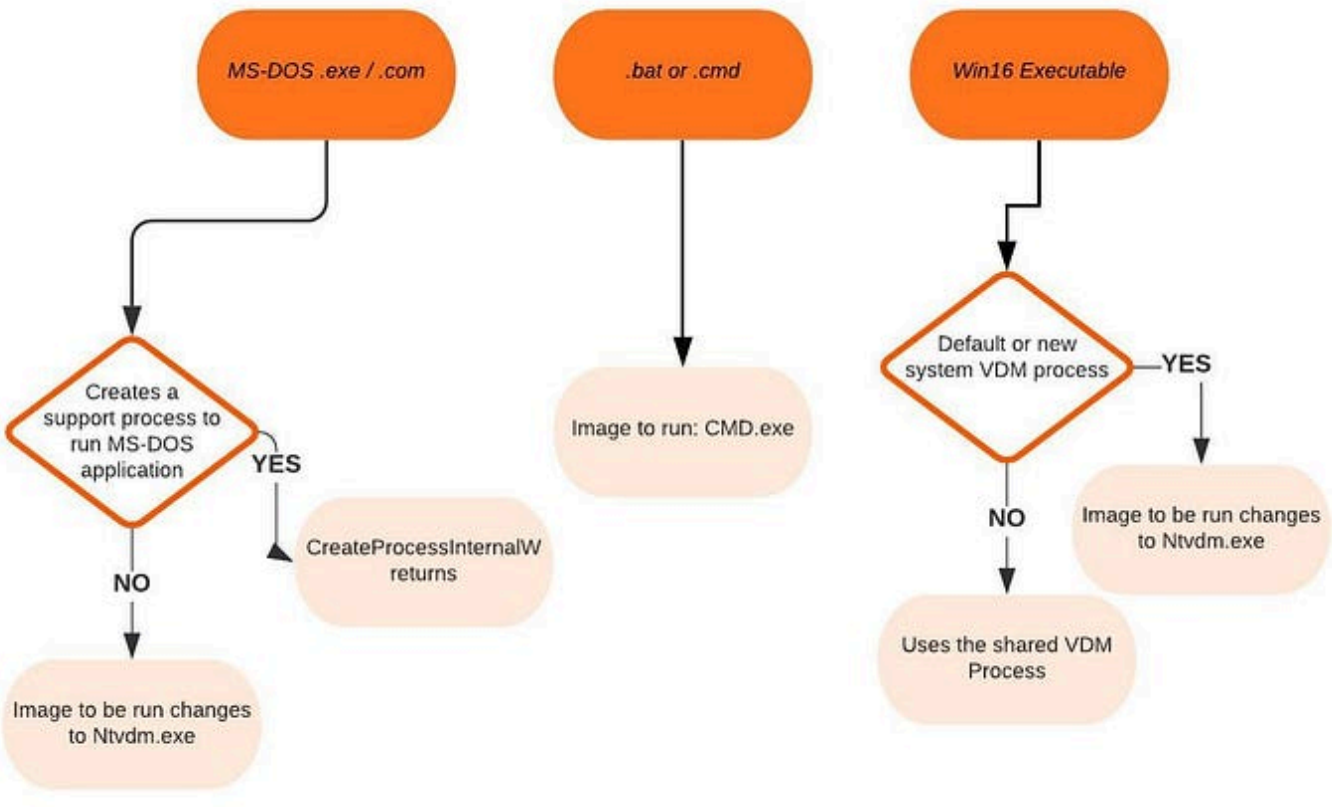
After completion of the previous steps, a call to `NtCreate-UserProcess` to attempt the creation of the process.

Stage 2: Opening the image to be executed.

Continuing the `NtCreateUserProcess` system call in the kernel-mode:

- `NtCreateUserProcess`first validates arguments and builds an internal structure to hold all creation information for validation and security intent.
- The second step involves choosing a Windows image to activate.1. DOS .bat or .cmd > Run cmd.exe2. Win16 > Run NtVdm.exe3. Windows > Run Exe directly4. DOS .exe, .com, or .pif

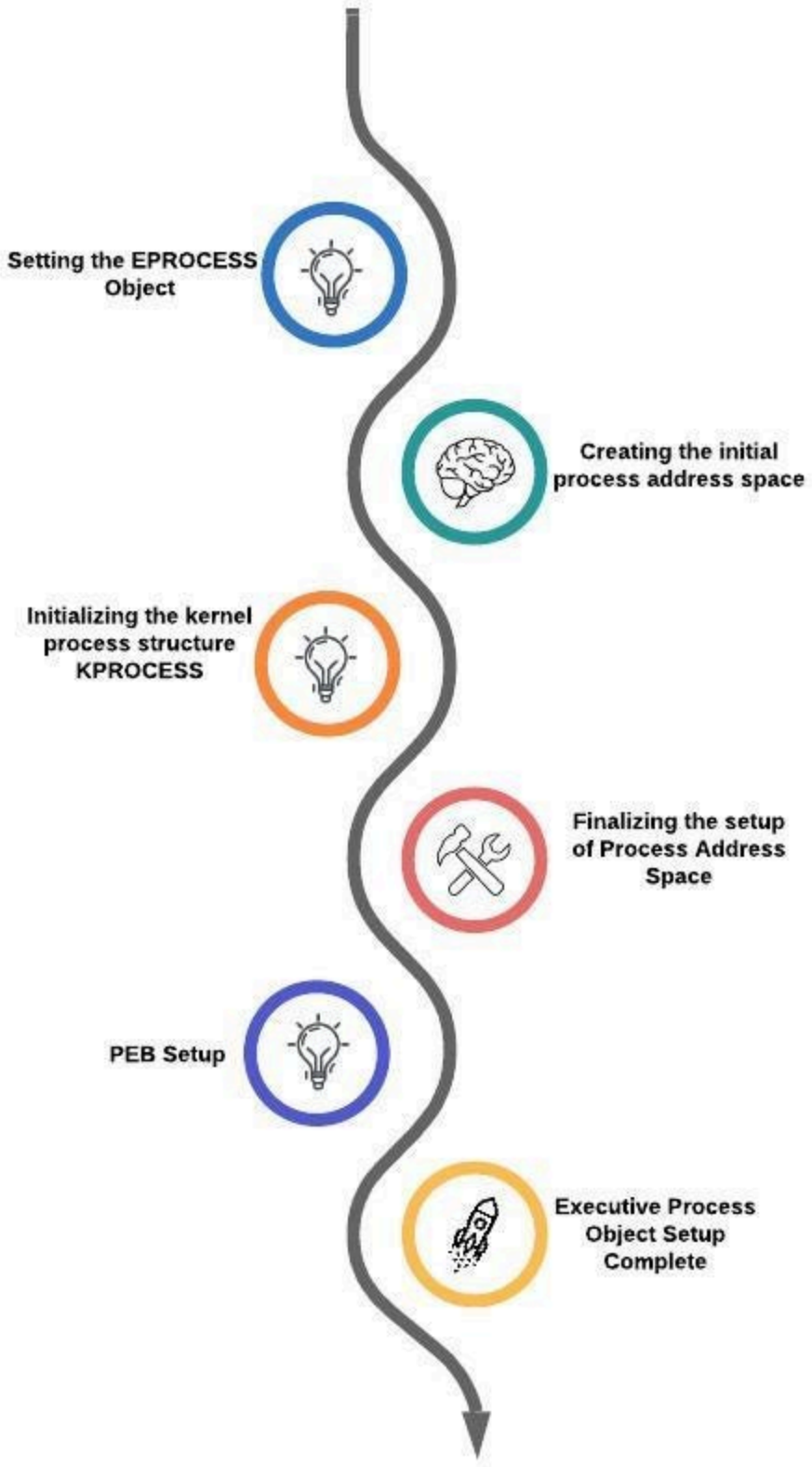
- If the process needs to be created protected, it also checks the signing policy.
- If the process to be created is modern, a licensing check is done to make sure it's licensed and allowed to run.
- If the process is a Trustlet (see part-1), the section object must be created with a special flag that allows a secure kernel to use it.
- If the executable file specified is a Windows EXE, `NtCreateUserProcess` tries to open the file and create a section object for it. The object isn't mapped into memory yet, but it is opened.
- `NtCreateUserProcess` looks in the registry under HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image FileExecution Options to see whether a subkey with the file name and extension of the executable image exists there. If it does, `PspAllocate-Process` look for a value named for that key.
- If the file specified is not a Windows EXE, `CreateProcessInternalW` tries to find a support image to run it.



CreateProcess Decision Matrix

Stage 3: Creating the Windows Executive Process Object

Next step, create a Windows Executive process object to run the image by calling the internal system function `PsPAllocateProcess` .



Stages of Executive Process Object Setup

Stage 3A: Setting up the EPROCESS object

1. Inherit the affinity of the parent process unless explicitly specified. Inherit the I/O and page priority from parent process /default: 5
2. Set the new process exit status to STATUS_PENDING. Choose the hard error process mode selected by the attribute list. Store the parent process' ID in the `InheritedFromUniqueProcessId` field in the new process object.
3. Query IFEO key to check whether the process should be mapped with large pages (!exception: WoW64 [WoW64 Auxillary Structure `EWOW64PROCESS`]). Query performance option key in IFEO: `PerfOptions` > `IoPriority, PagePriority, CpuPriorityClass, WorkingSetLimitInKB.`
4. If the process is to be created inside an AppContainer, validate that the token was created with a LowBox.
5. Attempt to acquire the necessary privileges, mapping the process with large pages, and creating the process within a new session with appropriate privilege. New processes inherit the security profile of their parents. If launched with a separate token, any change might happen only if the parent token's integrity level dominates the integrity level of the access token. (can be bypassed through `SeAssignPrimaryToken` privilege)
6. The session ID of the new process token is cross-referenced with the parent process and attached to the target session for processing.
7. Set the new process's quota block to the address of its parent process'squota block, and increment the reference count for the parent's quotablock. If the process was created through `CreateProcessAsUser`, thisstep won't occur. Instead, the default quota is created, or a quota matchingthe user's profile is selected.
8. The process minimum and maximum working set sizes are set to thevalues of `PspMinimumWorkingSet`and `PspMaximumWorkingSet`,respectively.
9. The next part involves the initialization of the address space of the process and detaching from the target session if it was different.
10. Initialize the KPROCESS part of the process object thus setting the token for the process. The process handle table is initialized. The final priority class and the default quantum for its threads are computed along with the various mitigation options provided in the IFEO key are read and set.

Stage 3B: Creating the initial process address space

The initial process address space consists of page directory, hyperspace page, VAD Bitmap page, working set list.

1. Page table entries are created in the appropriate page tables to map the initial pages.
2. The number of pages is deducted from the kernel variable `MmTotalCommittedPages` and added to `MmProcessCommit` .
3. The system-wide default process minimum working set size is deducted from `MmResidentAvailablePages` .
4. The page table pages for the global system space are created.

Stage 3C: Creating the kernel process structure

The next stage of `PspAllocateProcess` is the initialization of the KPROCESS structure. This task is executed by `KeInitializeProcess` which does the following:

1. The doubly linked list, which connects all threads part of the process is initialized.
2. The initial value of the process default quantum is hard-coded to 6 until initialized later
3. The process's base priority is set based on what was computed in stage 3A.
4. The default processor affinity for the threads in the process is set, as is the group affinity. The process-swapping state is set to resident following the selection of thread seed for the process.
5. If the process is a secure process, then its secure ID is created now by calling `HvlCreateSecureProcess.`

Stage 3D: Concluding the process address space setup

Handled mostly through `MmInitializeProcess-AddressSpace` {supports process cloning: yeah you little forkers}

1. The VMM sets the value of the process's last trim time to the current time.
2. The memory manager initializes the process's working set list. Page faults can be accounted for. The section is now mapped into the new process's address space and the process section base address is set to the base address of the image.
3. PEB is created and initialized. Ntdll.dll is mapped into the process. {32-bit Ntdll.dll for WoW64 processes} A new session, if requested, is now created for the process. The standard handles are duplicated and the new values are written in the process parameters structure. Any memory reservations listed in the attribute list are now processed. Additionally, two flags allow the bulk reservation of the first 1 or 16 MB of the address space.
4. The user process parameters are written into the process, copied, and fixed up. Affinity information is written into the PEB. The `MinWin` API redirection set is mapped into the process and its pointer is stored in the PEB.
5. Finally, the unique PID is determined and stored. Although, the kernel does not distinguish b/w the unique process and thread IDs and handles. The process and thread IDs are stored in the global handle table `PspCidTable` that is not associated with any process.
6. If the process is secure, the secure process is initialized and associated with the kernel process object.

Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Largest
Total	4,355,108 K	93,528 K	8,300 K	13,816 K	2,508 K	11,308 K	11,308 K		535	
Unusable	3,532 K									60 K
Stack	5,120 K	128 K	128 K	68 K	68 K				15	1,024 K
Shareable	25,452 K	3,108 K		208 K		208 K	208 K		22	20,484 K
Private Data	4,229,340 K	136 K	136 K	132 K	124 K	8 K	8 K		12	4,194,432 K
Page Table	356 K	356 K	356 K	356 K	356 K					
Mapped File	796 K	796 K		64 K		64 K	64 K		1	796 K
Managed Heap										
Image	88,512 K	88,512 K	7,252 K	12,556 K	1,532 K	11,024 K	11,024 K		474	8,756 K
Heap	2,000 K	492 K	428 K	432 K	428 K	4 K	4 K		11	1,024 K

Memory Snapshot for a process

Stage 3E: PEB Setup

1. `NtCreateUserProcess` calls `MmCreatePEB` which first maps the system-wide National Language Support tables into the process's address space.
2. Next, it calls `MiCreatePebOrTeb` to allocate a page for the PEB and then initializes a number of fields such as `MmHeap*` values, `MmCriticalSectionTimeout` and `MmMinimumStackCommitInBytes`.
3. If the `IMAGE_FILE_UP_SYSTEM_ONLY` flag is set, a single CPU is chosen for all the threads in this new process to run on. The selection process is performed by simply cycling through the available processors.

Stage 3F: Finalizing Executive Process Object setup

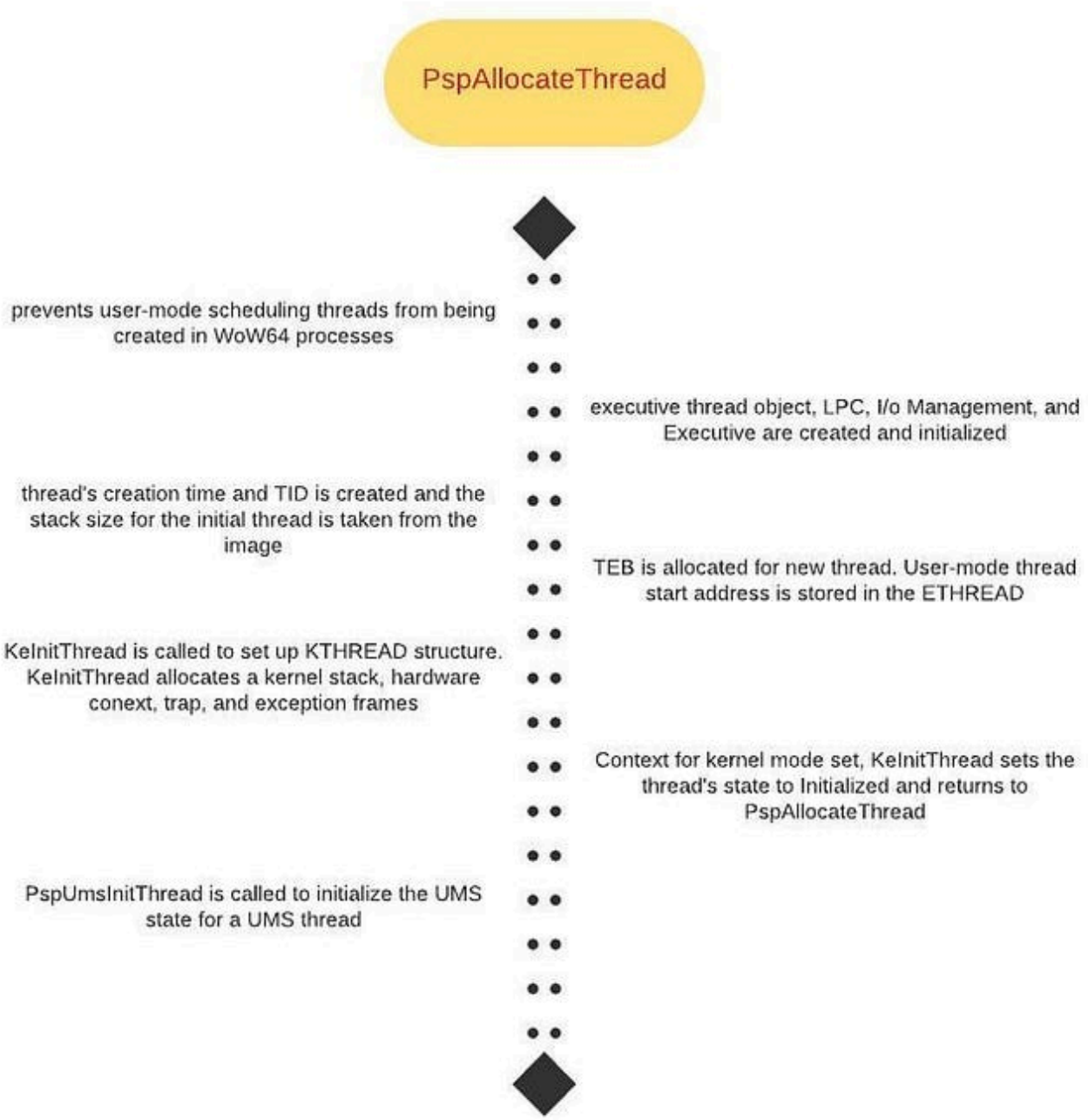
Before the handle to the new process can be returned, few final steps are performed by `PspInsertProcess` and its helper functions:

1. If system-wide auditing is enabled, the process's creation is written to the security event log.
2. If the parent process was contained in a job, the job is recovered from the job level set of the parent and then bound to the session of the newly created process.
3. The new process object is inserted at the end of the Windows list of active processes `PsActive-ProcessHead` which makes it accessible via functions like EnumProcesses and OpenProcess.
4. The process debug port of the parent process is copied to the new child process unless the `NoDebugInherit` flag is set.
5. Job objects can specify restrictions on which group or groups the threads within the processes part of a job can run on.
6. Finally, `PspInsertProcess` creates a handle for the new process by calling `ObOpenObjectByPointer` , and then returns this handle to the caller.

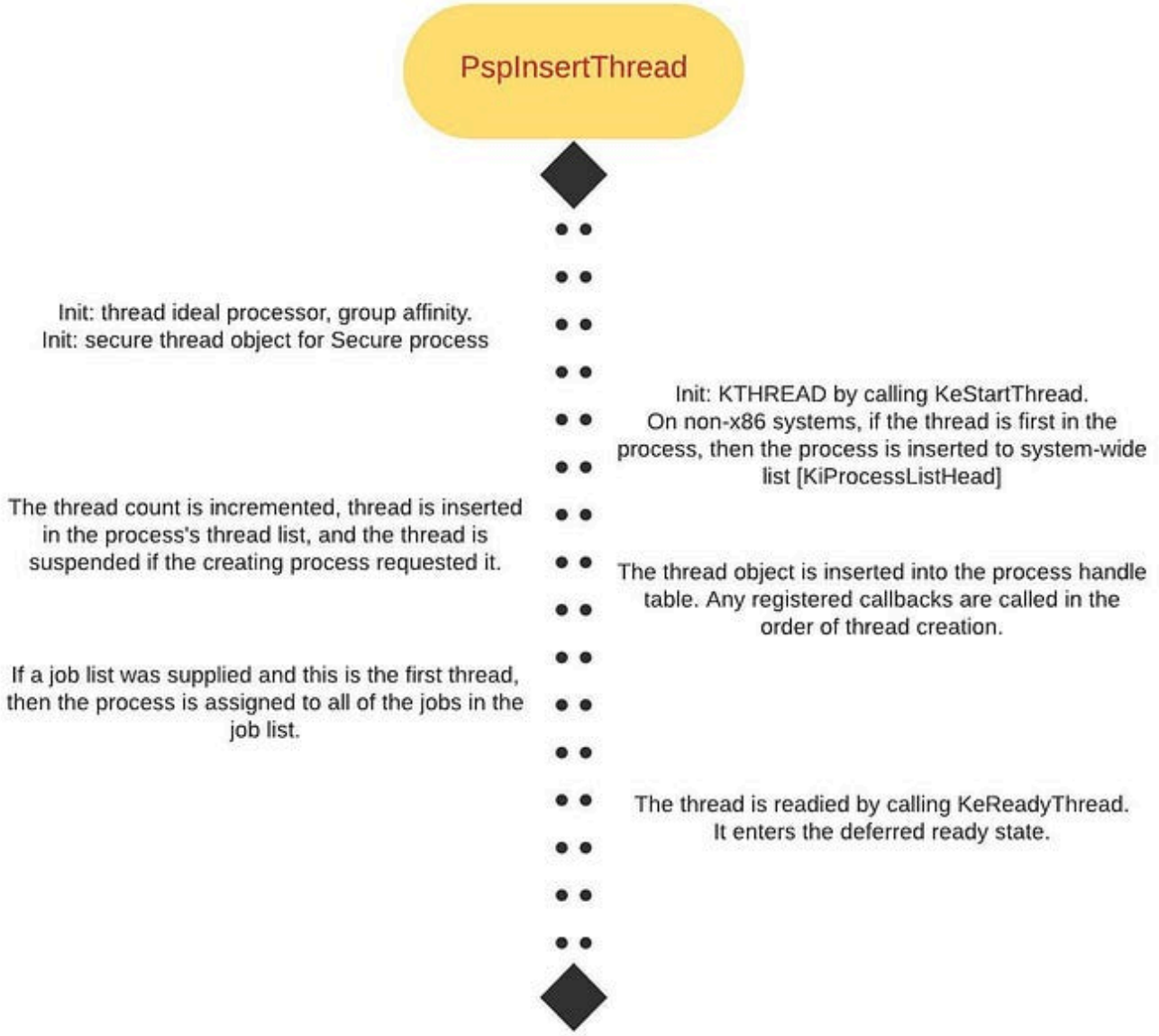
Stage 4: Creating the initial thread and its stack and context

`PspCreateThread` routine is responsible for all aspects of thread creation and is called by `NtCreateThread` when a new thread is being created. The helper routines `PspInsertThread` handle the actual creation and initialization of the executive thread object and `PspInsertThread`

handles the creation of the thread handle and security attributes and the call to `KeStartThread` to turn the executive object into a schedulable thread on the system. The following two charts explain the flow of `PspAllocateThread` and `PspInsertThread` .



PspAllocateThread flow



PspInsertThread flow

Stage 5: Performing Windows subsystem-specific initialization

Once `NtCreateUserProcess` returns with a success code, `CreateProcessInternalW` then performs Windows subsystem-specific operations to finish initializing the process. If software restriction policies dictate, a restricted token is created for the new process. Afterward, the application-compatibility database is queried to see whether an entry exists in either the registry or system application database for the process.

`CreateProcessInternalW` acquires SxS information such as manifest files and DLL redirection paths, and other flags. A message to the Windows subsystem is constructed based on the information collected to be sent to Csrss containing the following information:

Path name and SxS path name, handles: process, thread, section, access token, media information, AppCompatt and shim data, Immersive process information, PEB address, flags: protected? IsElevationRequired? WindowsApp?, UI language information, DLL redirection and .local flags, manifest file information.

Further, the windows subsystem performs the following steps:

1. `CsrCreateProcess` duplicates a handle for the process and thread. `CSR_PROCESS` structure is allocated.
2. The new process's exception port is set to be the general function port forthe Windows subsystem so that the Windows subsystem will receive amessage when a second-chance exception occurs in the process.
3. If a new process group is to be created with the new process serving as the root, then it's set in `CSR_PROCESS` .
4. The Csrss thread structure `CSR_THREAD` is allocated and initialized. `CsrCreateThread` inserts the thread in the list of threads for the process.
5. With an increase in the process count for the session, the process shutdown level is set to 0x280, the default process shutdown level. The new Csrss process structure is inserted into the list of Windows subsystem-wide processes.

Stage 6: Starting execution of the initial thread

At this point, the process environment has been determined, resources for its threads to use have been allocated, the process has a thread, and the Windows subsystem knows about the new process. Unless the caller specified the `CREATE_SUSPENDED`flag, the initial thread is now resumed so that it can start running and perform the remainder of the process-initialization work that occurs in the context of the new process.

Stage 7: Performing process initialization in the context of the new process

The new thread begins life running the kernel-mode thread startup routine `KiStartUserThread`. `KiStartUserThread`lowers the thread's IRQL level from deferred procedure call (DPC) level to APC level and then calls the system initial thread routine, `PspUserThreadStartup` . The user-specified thread start address is passed as a parameter to this routine. `PspUserThreadStartup`performs the following actions:

1. It installs an exception chain on x86 architecture. It lowers IRQL to `PASSIVE_LEVEL` (0). It disables the ability to swap the primary access token at runtime.
2. If the thread was killed on startup, it's terminated and no further action is taken.

3. It sets the locale ID and ideal processor in the TEB. It calls `DbgkCreateThread` , which checked whether image notifications were sent for the new process. If they weren't, and notifications are enabled, an image notification is sent first for the process and then for the image load of Ntdll.dll. {since no PID (required for kernel callouts) allocated at that time}
4. If the process is a debugger, then a create process message is sent through the debug object so that the process startup debug event can be sent to the appropriate debugger process. (CREATE_PROCESS_DEBUG_INFO)
5. It checks whether application prefetching is enabled on the system and, if so, calls the prefetcher to process the prefetch instruction file and prefetch pages referenced during the first 10 seconds the last time the process ran.
6. It checks whether the system-wide cookie in the `SharedUserData` structure has been set up. If it hasn't, it generates it based on a hash of system information such as the number of interrupts processed, DPC deliveries, page faults, interrupt time, and a random number. This systemwide cookie is used in the internal decoding and encoding of pointers, such as in the heap manager to protect against certain classes of exploitation.
7. If the process is secure, then a call is made to `HvlStartSecureThread` that transfers control to the secure kernel to start thread execution. This function only returns when the thread exits.
8. It sets up the initial thunk context to run the image-loader initialization routine `LdrInitialize-Thunk` in Ntdll.dll, as well as the system-wide thread startup stub `RtlUserThreadStart` in Ntdll.dll. The `LdrInitializeThunk` routine initializes the loader, the heap manager, NLS tables, thread-local storage (TLS) and fiber-local storage (FLS) arrays, and critical section structures. It then loads any required DLLs and calls the DLL entry points with the `DLL_PROCESS_ATTACH`function code.

Once the above function returns, `NtContinue` restores the new user context and returns to user mode. Thread execution now finally starts. `RtlUserThreadStart` uses the address of the actual image entry point and the start parameter and calls the application's entry point. These two parameters have also already been pushed onto the stack by the kernel.

This complicated series of events has two purposes:

- It allows the image loader inside Ntdll.dll to set up the process internally and behind the scenes so that other user-mode code can run properly.
- Having all threads begin in a common routine allows them to be wrappedin exception handling so that if they crash, Ntdll.dll is aware of that andcan call the unhandled exception filter inside Kernel32.dll. It is also able tocoordinate thread exit on return from the thread's start routine and toperform various cleanup work.

This concludes the creation and startup of a Windows process.

References:

1. https://docs.microsoft.com/en-us/
2. Windows Internals 7th Edition, Part 1
3. 8.8.8.8