OPERATING SYSTEMS LAB

COE-319

# macOS and Windows : A Comparative Study

Threads

Processes Management

Memory Management

Deadlock Handling

Swapnil Dhanwal - 368CO14

Vishnu Bhaskar - 389CO14

Vibhor Bijoy - 384CO14

COE-3

Threads in MacOS:

MacOS has a preemptive thread mechanism that is robust and scalable.

Each process in macOS is made up of one or more threads, each of which represents a single path of execution through the application’s code. Each application starts with a single thread, which runs in the application’s *main* function.

When an application spawns a new thread, that thread becomes an independent entity inside of the application’s process space. Each thread has its own execution stack and is scheduled for runtime separation through the kernel. A macOS thread can communicate with other threads and processes, perform I/O operations and more. All threads of the same application share the same virtual memory space and the same access rights as the application itself.

Applications can spawn new threads using several methods, as described below:

NSThread Class: In Apple’s proprietary programming languages, Objective-C and Swift, new threads can be created using the NSThread class. Each thread has a function or a method as an entry point. They can be created by the following method:

let thread = NSThread(target : self, selector : #selector(viewController.doSomething), object : nil)

func doSomething() -> Void

{

//do something

}

thread.start()

The first line is the initialisation. The function doSomething serves as the entry point for the thread and its contents will be executed when the thread is run. The third line actually starts the thread.

Threads can also be created using the POSIX Thread library of C.

#include <pthread>

void \*posixThreadRoutine()

{

//do something

}

int launchThread()  
{

pthread\_attr\_t attr;

pthread\_t posixThreadId;

int returnVal;

returnVal = pthread\_attr\_init(&attr);

returnVal = pthread\_attr\_setdetachstate(&posixThreadID, &attr, &posixThreadRoutine, PTHREAD\_CREATE\_DETACHED);

}

Threads in Windows:

In Windows, same as macOS, a thread is the basic uint to which the operating system allocates processor time. A thread can execute as part of the process code, including parts currently being executed by another thread. Microsoft Windows supports preemptive multitasking, which creates the effect of simultaneous execution of multiple threads from multiple processes. On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer.

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

An *application* can use the thread pool to reduce the number of application threads and provide management of the worker threads. Applications can queue work items, associate work with waitable handles, automatically queue based on a timer, and bind with I/O.

*User-mode scheduling* (UMS) is a lightweight mechanism that applications can use to schedule their own threads. An application can switch between UMS threads in user mode without involving the [system scheduler](https://msdn.microsoft.com/en-us/library/windows/desktop/ms685096(v=vs.85).aspx) and regain control of the processor if a UMS thread blocks in the kernel. Each UMS thread has its own thread context instead of sharing the thread context of a single thread. The ability to switch between threads in user mode makes UMS more efficient than thread pools for short-duration work items that require few system calls.

Threads can be created using the following method in Windows:

The CreateThread function creates a new thread for a process. The creating thread must specify the starting address of the code that the new thread is to execute. Typically, the starting address is the address of a function defined in the program code.

#include <windows.h>

DWORD WINAPI ThreadFunc(void\* data)

{

// Do stuff. This will be the first function called on the new thread.

// When this function returns, the thread goes away. See MSDN for more details.

return 0;

}

int main()

{

HANDLE thread = CreateThread(NULL, 0, ThreadFunc, NULL, 0, NULL);

if (thread)

{

// Optionally do stuff, such as wait on the thread.

}

}

Processes in Windows

A windows process contains its own independent virtual address space with both code and data, protected from other processes. Each process, in turn, contains one or more independently executing threads. By creating and managing processes, applications can have multiple, concurrent tasks processing files, performing calculations, or communicating with other networked systems. It is even possible to improve application performance by exploiting multiple CPUs.

Each windows process control block (PCB) contains the includes the following resources:

1. One or more threads
2. A virtual address space that is distinct from other processes’ address spaces.
3. One or more code segments
4. One or more data segments, containing global variables
5. Environment strings with environment variable information such as the current search path
6. The process heap

Process creation:

BOOL CreateProcess (

LPCTSTR lpApplicationName,

LPTSTR lpCommandLine,

LPSECURITY\_ATTRIBUTES lpsaProcess,

LPSECURITY\_ATTRIBUTES lpsaThread,

BOOL bInheritHandles,

DWORD dwCreationFlags,

LPVOID lpEnvironment,

LPCTSTR lpCurDir,

LPSTARTUPINFO lpStartupInfo,

LPPROCESS\_INFORMATION lpProcInfo);

1. *lpApplicationName* and *lpCommandLine* (this is an LPTSTR and not an LPCTSTR) together specify the executable program and the command line arguments
2. *lpsaProcess* and *lpsaThread* point to the process and thread security attribute structures.
3. *bInheritHandles* indicates whether the new process inherits copies of the calling process's inheritable open handles (files, mappings, and so on).
4. *dwCreationFlags* combines several flags : CREATE\_SUSPENDED, DETACHED\_PROCESS and CREATE\_NEW\_CONSOLE, CREATE\_NEW\_PROCESS\_GROUP

There is a marked different between process models in Windows and UNIX based operating systems (such as macOS). Windows, firstly, has no equivalent of the *fork* system call - it is rather difficult to emulate *fork* in Windows. *Fork* is difficult to use in a multithreaded environment because it has to create a copy of all threads and synchronisation objects. Additionally, Windows does not contain any parent-child process relationships. Windows processes are identified by both IDs and handles whereas macOS processes require no handles for the same.

Process scheduling:

Old versions of Windows, such as Win XP schedule using a priority-based scheduler that uses a *round-robin* methodology. The scheduler ensures that the highest level priority thread/process will always run first. When a process returns from the waiting stage, it receives a priority boost.

When a processor becomes available, the system performs a context switch. The steps in a context switch are:

1. *Save the context of the thread that just finished executing.*
2. *Place the thread that just finished executing at the end of the queue for its priority.*
3. *Find the highest priority queue that contains ready threads.*
4. *Remove the thread at the head of the queue, load its context, and execute it.*

PROCESSES IN MACOS

In macOS, the process control block (PCB) contains information such as the current state of the process, the address size of its partition, its type, its creator, a copy of all process-specific global variables, information about its size and a process serial number. The process information is referred to as the *context* of the process. The process manager assigns a process serial number to identify each process. Two types of processes exist in the paradigm of CPU access - foreground processes and background processes. A foreground process has primacy in accessing the CPU for computation tasks. Applications can be designed without user interfaces, constituting background processes.

Process creation:

The application jump table contains one entry for every externally referenced routine in every code segment of your application.

When you create an application, you specify in its 'SIZE' resource how much memory you want the Process Manager to allocate for your application’s partition. You specify two values: the preferred amount of memory to allocate and the minimum amount of memory to allocate. When a user opens your application from the Finder, the Process Manager first attempts to allocate a partition of the preferred size. If your application cannot be launched in the preferred amount of memory, the Finder might display a dialog box giving the user the option of opening the application using less than the preferred size. The Finder will not launch your application if the minimum amount of memory specified for your application is not available.

The Process Manager assigns the application a process serial number, records its context, and returns control to the launching application (usually the Finder). The Process Manager typically transfers control to the new application after the launching application *makesasubsequentcalltoWaitNextEvent* *orEventAvail*.

Process scheduling:

When your application is the foreground process, it yields time to other processes in these situations: when the user wants to switch to another application or when no events are pending for your application. Your application can also choose to yield processing time to other processes when it is performing a lengthy operation.

A major switch occurs when the Process Manager switches the context of the foreground process with the context of a background process (including the A5 worlds and application-specific system global variables) and brings the background process to the front, sending the previous foreground process to the background.

A minor switch occurs when the Process Manager switches the context of a process to give time to a background process without bringing the background process to the front.

info@mindfiresolutions.com Page 4 of 33

Mindfire White Paper

A background process should not perform any task that significantly limits the ability of the foreground process to respond quickly to the user.

Memory management WINDOWS 10

Each process on 32-bit & 64 bit Microsoft Windows has its own virtual address space that enables addressing up to 4 gigabytes & 8 terabytes of memory respectively. All threads of a process can access its virtual address space. The virtual address space for a process is the set of virtual memory addresses that it can use. The address space for each process is private and cannot be accessed by other processes unless it is shared.

A virtual address does not represent the actual physical location of an object in memory; instead, the system maintains a page table for each process, which is an internal data structure used to translate virtual addresses into their corresponding physical addresses. Each time a thread references an address, the system translates the virtual address to a physical address. If the threads of a process attempt to use more physical memory than is currently available, the system pages some of the memory contents to disk. The total amount of virtual address space available to a process is limited by physical memory and the free space on disk available for the paging file.

Physical storage and the virtual address space of each process are organized into pages, units of memory, whose size depends on the host computer. For example, on x86 computers the host page size is 4 kilobytes.

The pages of a process's virtual address space can be in one of the following states:

|  |  |
| --- | --- |
| State | Description |
| Free | The page is neither committed nor reserved. The page is not accessible to the process. It is available to be reserved, committed, or simultaneously reserved and committed. Attempting to read from or write to a free page results in an access violation exception. |
| Reserved | The page has been reserved for future use. The range of addresses cannot be used by other allocation functions. The page is not accessible and has no physical storage associated with it. It is available to be committed. |
| Committed | Memory charges have been allocated from the overall size of RAM and paging files on disk. The page is accessible and access is controlled by one of the [memory protection constants](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366786(v=vs.85).aspx). The system initializes and loads each committed page into physical memory only during the first attempt to read or write to that page. When the process terminates, the system releases the storage for committed pages. |

However, threads cannot access memory that belongs to another process, which protects a process from being corrupted by another process.

The memory manager creates the following memory pools that the Windows uses to allocate memory: nonpaged pool and paged pool. Both memory pools are located in the region of the address space that is reserved for the system and mapped into the virtual address space of each process. The nonpaged pool consists of virtual memory addresses that are guaranteed to reside in physical memory as long as the corresponding kernel objects are allocated. The paged pool consists of virtual memory that can be paged in and out of the system.

The handles for [kernel objects](https://msdn.microsoft.com/en-us/library/windows/desktop/ms724485(v=vs.85).aspx) are stored in the paged pool, so the number of handles you can create is based on available memory. The system records the limits and current values for its nonpaged pool, paged pool, and page file usage.

Memory that belongs to a process is implicitly protected by its private virtual address space. In addition, Windows provides memory protection by using the virtual memory hardware. The implementation of this protection varies with the processor, for example, code pages in the address space of a process can be marked read-only and protected from modification by user-mode threads.

Copy-on-write protection is an optimization that allows multiple processes to map their virtual address spaces such that they share a physical page until one of the processes modifies the page. This is part of a technique called lazy evaluation, which allows the system to conserve physical memory and time by not performing an operation until absolutely necessary.

Memory management in macOS

Mach supplies large portions of virtual address spaces for addressing. At initial access time, the virtual address may not correspond to a location in physical memory. Mach’s task is to connect each virtual address to its physical space in memory. This is done through demand paging. Data in an address space is made up of memory objects, which are further broken down into pages. When connecting a virtual address to a location in physical memory, Mach asks the owner of a memory object for the contents of a page, and returns the data (which may have been modified) to the owner. Another name for the owner of a memory object is a pager. Pagers provide the data of the pages in memory objects. There are two built-in pagers in the Mac OS X, the default pager and the vnode pager.

The default pager handles anonymous memory. Anonymous memory is virtual memory that is nonpersistent – it exists only as long as the task exists.

The vnode pager maps files into memory objects. This is done through an interface that Mach exports to memory objects to allow their contents to be contributed by user-mode tasks. This interface is called the External Memory Management Interface (EMMI). EMMI gives handles to virtual memory, called named memory entries. By having names attached to memory, the owner can map the entry or allow others to do so. Two different tasks may map a single entry; this results in a shared memory window between them. This allows flexibility in sharing memory.

Additional methods for populating address space are direct allocation and inheritance. In direct allocation, the virtual memory object is anonymous and supported by the default pager. In inheritance, shared address space is established as new tasks are cloned from a parent. Memory address space is also cloned, and mapped memory objects may be inherited as a copy, shared, or not inherited, depending on the mapping.

The kernel maintains and queries three system-wide lists of physical memory pages:

The active list contains pages that are currently mapped into memory and have been recently accessed.

The inactive list contains pages that are currently resident in physical memory but have not been accessed recently. These pages contain valid data but may be removed from memory at any time.

The free list contains pages of physical memory that are not associated with any address space of VM object. These pages are available for immediate use by any process that needs them.

When the number of pages on the free list falls below a threshold (determined by the size of physical memory), the pager attempts to balance the queues. It does this by pulling pages from the inactive list. If a page has been accessed recently, it is reactivated and placed on the end of the active list.

In order to optimize the performance of inherited copies, Mach uses copy-on-write, a delayed form of copying. This is accomplished by protected sharing. Instead of directly copying the range of addresses, the two tasks share the memory being copied with read only access. A portion is copied when either tasks modifies a part of the memory.

Deadlock handling in windows:

Windows provides support for deadlock detection, but not avoidance and prevention. This is based on the Ostrich algorithm, according to which, if the probability of a problem occurring is very low, then there is no need to install avoidance or prevention measures - it is easier to simply start over.

Deadlock Detection monitors the driver's use of resources which need to be locked -- spin locks, mutexes, and fast mutexes. This Driver Verifier option will detect code logic that has the potential to cause a deadlock at some future point. Driver Verifier's Deadlock Detection routines find lock hierarchy violations that are not necessarily simultaneous. Most of the time, these violations identify code paths that will deadlock when given the chance.

To find potential deadlocks, Driver Verifier builds a graph of resource acquisition order and checks for loops. If you were to create a node for each resource, and draw an arrow any time one lock is acquired before another, then path loops would represent lock hierarchy violations.

Driver Verifier will issue a bug check when one of these violations is discovered. This will happen before any actual deadlocks occur. When Deadlock Detection finds a violation, it will issue bug check 0xC4. The first parameter of this bug check will indicate the exact violation. Possible violations include:

1. Two or more threads involved in a lock hierarchy violation
2. A resource that is released out of sequence
3. A thread that tries to acquire the same resource twice (a self-deadlock)
4. A resource that is released without having been acquired first
5. A resource that is released by a different thread than the one that acquired it
6. A resource that is initialized more than once, or not initialized at all
7. A thread that is deleted while still owning resources

Driver Verifier can predict possible deadlocks. For example, trying to use the same KSPIN\_LOCK data structure both as a regular spin lock and as a stack queued spin lock.

Once Deadlock Detection finds a violation, the !deadlock kernel debugger extension can be used to investigate exactly what has occurred. It can display the lock hierarchy topology as well as the call stacks for each thread at the time the locks were originally acquired.

Deadlock handling in macOS

Deadlock prevention takes on a new look with the new OS X. Again, with UNIX as its basis, the OS uses cooperative multitasking. This type of multitasking allows any number of processes to be going on at one time but the OS decides which of these processes gets the processor power. If one program were to crash or hang up the processor the new OS X would, like UNIX, not crash but rather close out that process and flush the memory. Some believe that this is a weaker solution to multitasking than preemptive multitasking that is seen in Windows NT and the like. However, this type of tasking does not crash the system as often when one program/process experience a problem.