Sistemas Operativos

Threads - II

Sistemas Operativos

- ☑ Versión: Marzo 2018
- ☑ Palabras Claves: Threads, Hilos, ULT, KLT, Procesos, Concurrencia, Paralelismo, Multithreading, Linux,Solaris, Windows, Fibras, LWP, POSIX; clone

Algunas diapositivas han sido extraídas de las ofrecidas para docentes desde el libro de Stallings (Sistemas Operativos) y el de Silberschatz (Operating Systems Concepts)

Ejemplo - Linux (< 2.4)

- No considera el concepto de thread
- Es posible “clonar” un proceso para compartir recursos (archivos, memoria, etc.)
  - System Call clone()
- Mismo descriptor de proceso para “procesos” y “threads”

Ejemplo - Linux (< 2.4)

- System Call clone()
  - Versión modificada de fork()
  - Permite especificar que recursos compartir con su “tareas hijas”

```
int clone(  int (*fn) (void *arg),
            void *child_stack,
            int flags,
            void *arg)
```

Ejemplo - Linux (< 2.4)

- Posibles Flags:
  - CLONE_VM: Compartir espacio de memoria
  - CLONE_FS: Compartir información del File System (raiz del FS, dir. de trabajo, umask)
  - CLONE_FILES: Compartir la tabla de descriptores de archivos.
  - CLONE_SIGHAND: Compartir la tabla de manejadores de señal.

Ejemplo - Linux (> = 2.6)

- POSIX (Portable Operating System Interface)
  - Standart de System Calls (IEEE)
- Adopta el modelo NPTL (Native POSIX Threads Library)
  - Esquema 1:1
  - Threads y procesos utilizan la misma estructura de datos (task_struct)
  - Cada Threads de un proceso:
    - Tiene su propio PID
    - Comparten un TPID
    - La syscall getpid() retorna el TPID

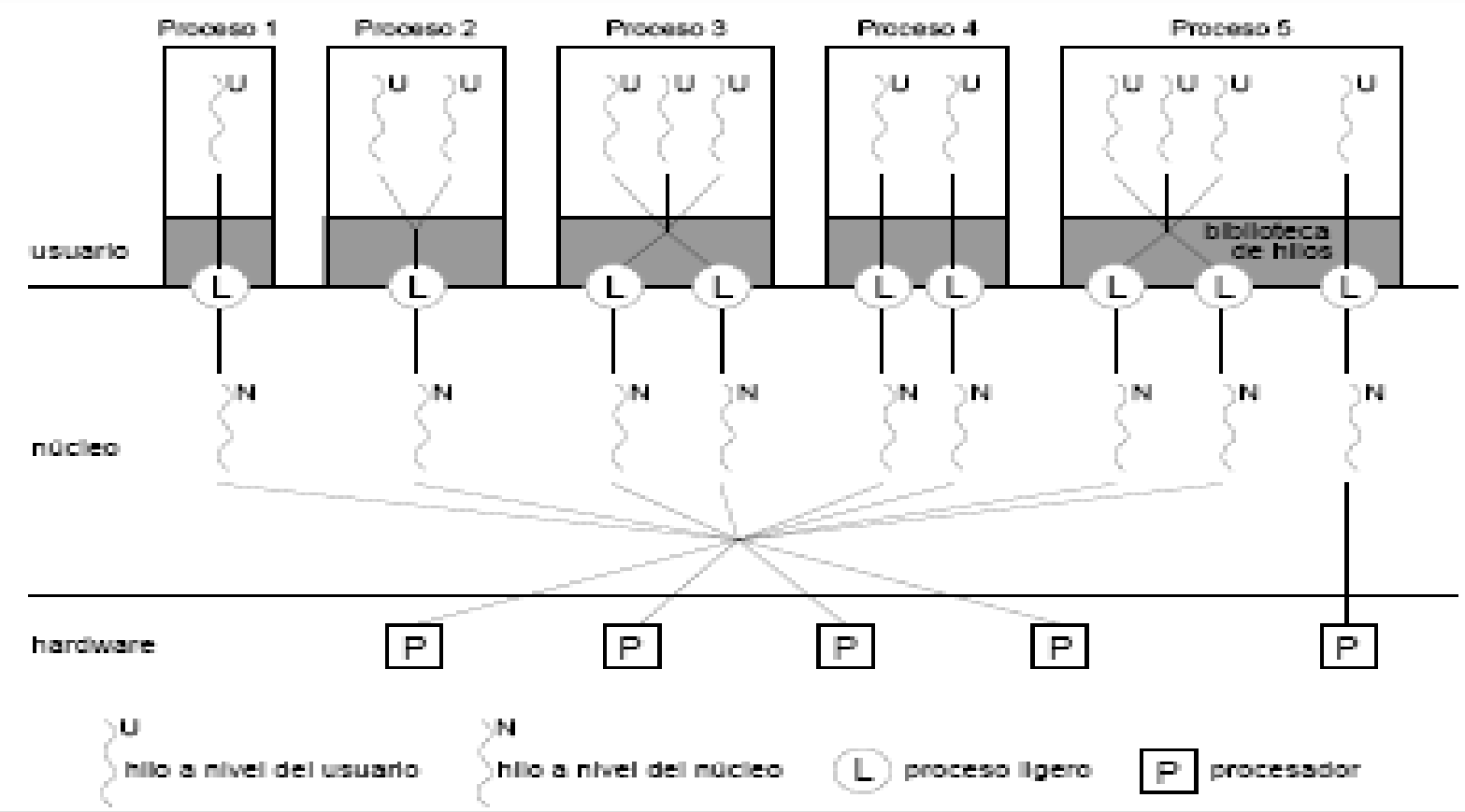
Linux

- Mas Información
  - http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library (04/2018)
  - https://computing.llnl.gov/tutorials/pthreads/ (04/2018)
  - http://en.wikipedia.org/wiki/POSIX_Threads (04/2018)
  - http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/Accessability.html (04/2018)

Ejemplo - Solaris

- Maneja un esquema combinado de ULT y KLT
- Considera 3 tipos de Thread:
  - ULT: Invisibles al Kernel
  - KLT: Planificación en el/los procesador/es
  - LWP
    - Correspondencia entre ULT y KLT.
    - Soporta 1 o mas ULT
    - Se corresponde con un KLT
    - Planificados independientemente por el Kernel

Ejemplo - Solaris

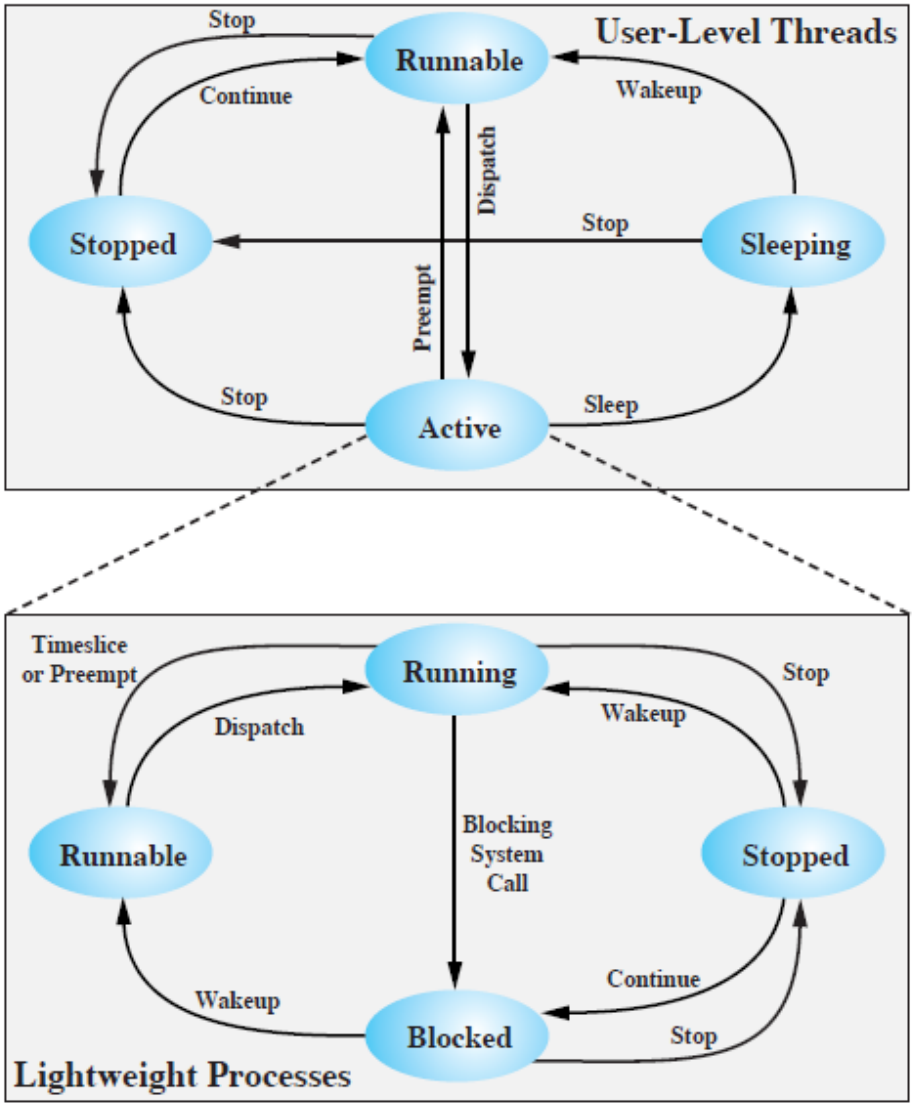


Ejemplo - Solaris

- Los ULT pueden estar
  - Ligados
    - Asociado permanentemente a un LWP
    - Aplicaciones en Tiempo Real
  - No ligados
    - No están permanentemente asociados a un LWP
    - Múltiplexación entre LWPs disponibles

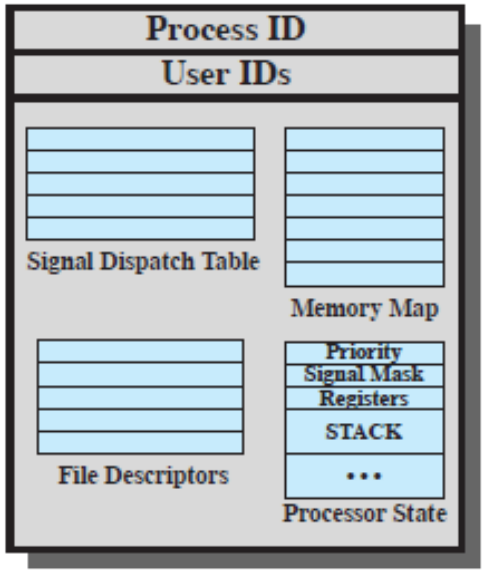
Ejemplo - Solaris

- Estructuras
  - ULT: identificador del hilo, el conjunto de registros del usuario, la pila y la prioridad.
  - LWP: conjunto de registros del usuario, la prioridad, la pila y la referencia al hilo del núcleo que lo soporta.
  - KLT: los registros del núcleo, un apuntador al LWP, e información sobre la prioridad y la planificación.

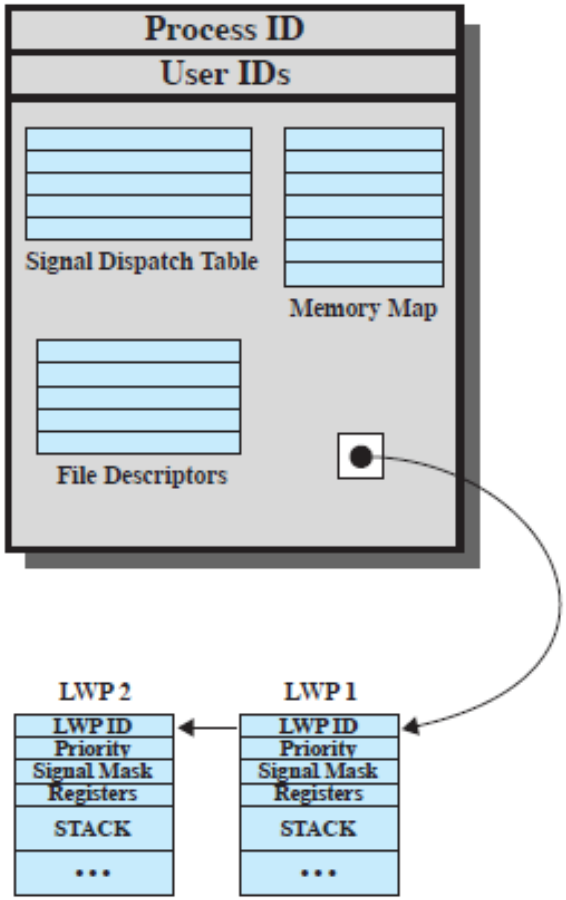

User-Level Threads
Runnable
Stop
Continue
Wakeup
Dispatch
Stopped
Stop
Sleeping
Preempt
Stop
Active
Sleep
Timeslice
or Preempt
Running
Stop
Dispatch
Wakeup
Runnable
Blocking
System
Call
Stopped
Wakeup
Continue
Blocked
Stop
Lightweight Processes

Ejemplo - Solaris

- **UNIX Process Structure**



- **Solaris Process Structure**



Ejemplo - Solaris

- Analizar:
  - Supongamos que se quiere trabajar con 5 archivos y hay 5 read que deben ocurrir simultáneamente.
  - Debe haber 5 LWP donde cada ULT haga el request para que se canalicen como requerimiento de I/O simultáneos que puedan llevarse a distintas CPUs (ideal: que esten en distintos discos...).
  - Si fueran 4 LWP, el 5to debería esperar el retorno de uno de los LWP para ejecutarse.

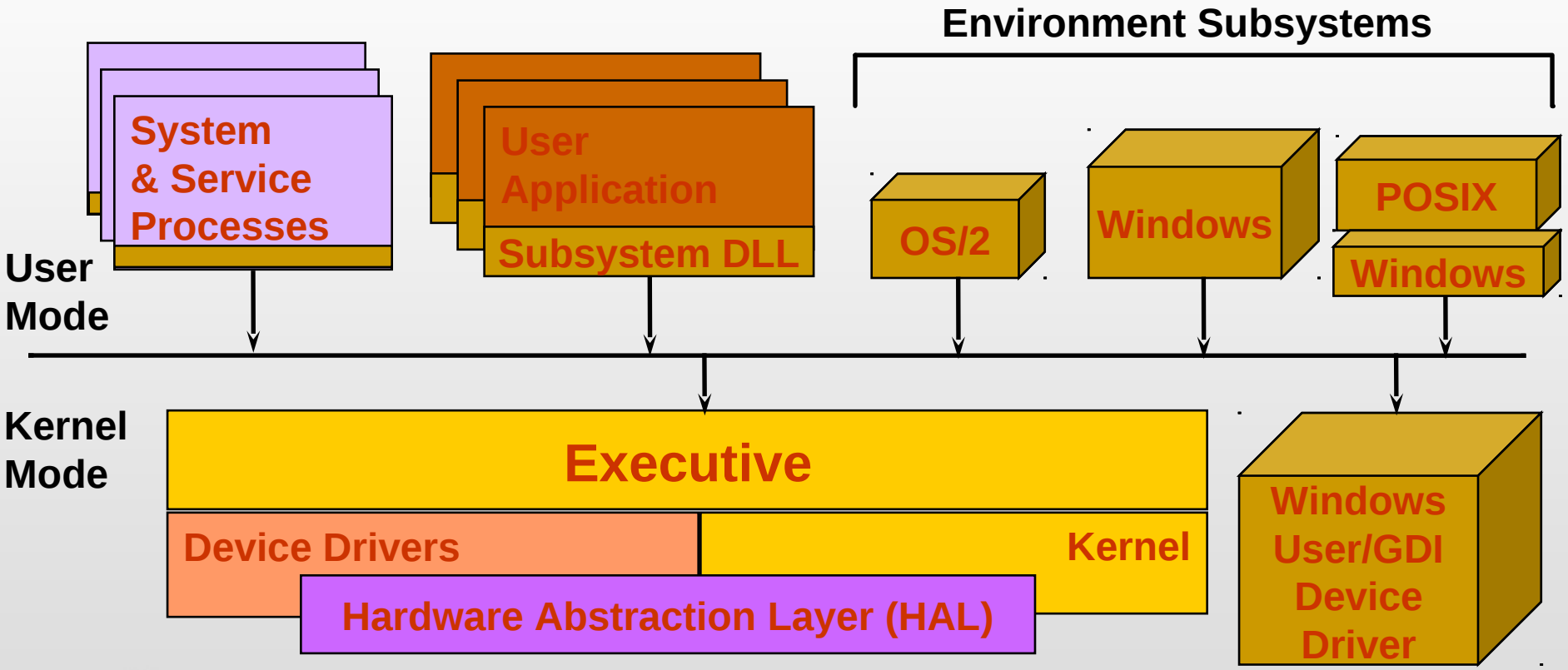
Solaris

- Mas Información:
  - http://www.cs.cf.ac.uk/Dave/C/node29.html (04/2018)

Ejemplo - Windows

- Modelo de Multithreading: Uno a Uno
  - Posible modelo Muchos a Uno utilizando Fibers (Fibras)
- Cada thread contiene:
  - Identificación
  - Contexto y Stacks (usuario y kernel)
  - Área de datos propia
- La creación involucra una System Call ubicada en “Kernel32.dll”

Organización



Executive

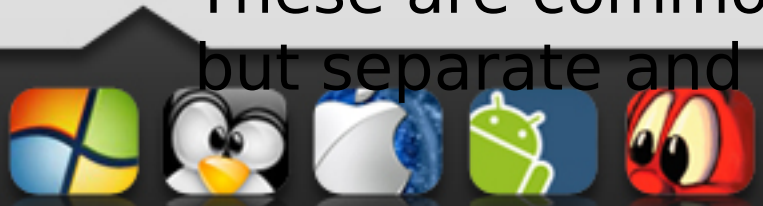
- Upper layer of the operating system
- Provides “generic operating system” functions (“services”)
  - Process Manager
  - Object Manager
  - Cache Manager
  - LPC (local procedure call) Facility
  - Configuration Manager
  - Memory Manager
  - Security Reference Monitor
  - I/O Manager
  - Power Manager
  - Plug-and-Play Manager
- Almost completely portable C code
- Runs in kernel (“privileged”, ring 0) mode
- Most interfaces to executive services not documented

Kernel

- Lower layers of the operating system
  - Implements processor-dependent functions (x86 vs. Itanium etc.)
  - Also implements many processor-independent functions that are closely associated with processor-dependent functions
- Main services
  - Thread waiting, scheduling & context switching
  - Exception and interrupt dispatching
  - Operating system synchronization primitives (different for MP vs. UP)
  - A few of these are exposed to user mode
- Not a classic “microkernel”
  - shares address space with rest of kernel-mode components

Per-Process Data

- Each process has its own…
  - Virtual address space (including program code, global storage, heap storage, threads' stacks)
  - processes cannot corrupt each other's address space by mistake
  - Working set (physical memory "owned" by the process)
  - Access token (includes security identifiers)
  - Handle table for Windows kernel objects
  - Environment strings
  - Command line
  - These are common to all threads in the process, but separate and protected between processes



Per-Thread Data

- Each thread has its own…
  - User-mode stack (arguments passed to thread, automatic storage, call frames, etc.)
  - Kernel-mode stack (for system calls)
  - Thread Local Storage (TLS) – array of pointers to allocate unique data
  - Scheduling state (Wait, Ready, Running, etc.) and priority
  - Hardware context (saved in CONTEXT structure if not running)
    - Program counter, stack pointer, register values
    - Current access mode (user mode or kernel mode)
  - Access token (optional -- overrides process's if present)

Process Windows APIs

- ☑ CreateProcess
- ☑ OpenProcess
- ☑ GetCurrentProcessId - returns a global ID
- ☑ GetCurrentProcess - returns a handle
- ☑ ExitProcess
- ☑ TerminateProcess - no DLL notification
- ☑ Get/SetProcessShutdownParameters
- ☑ GetExitCodeProcess
- ☑ GetProcessTimes
- ☑ GetStartupInfo

Windows Thread APIs

- CreateThread
- CreateRemoteThread
- GetCurrentThreadId - returns global ID
- GetCurrentThread - returns handle
- SuspendThread/ResumeThread
- ExitThread
- TerminateThread - no DLL notification
- GetExitCodeThread
- GetThreadTimes
- Windows 2000 adds:
  - OpenThread
  - new thread pooling APIs

Process Creation

- ☑ No parent/child relation in Win32
- ☑ *CreateProcess()* – new process with primary thread

```
BOOL CreateProcess(
    LPCSTR lpApplicationName,
    LPSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation)
```

UNIX & Win32 comparison

- Windows API has no equivalent to fork()
- CreateProcess() similar to fork()/exec()
- UNIX $PATH vs. lpCommandLine argument
  - Win32 searches in dir of curr. Proc. Image; in curr. Dir.; in Windows system dir. (GetSystemDirectory); in Windows dir. (GetWindowsDirectory); in dir. Given in PATH
- Windows API has no parent/child relations for processes
- No UNIX process groups in Windows API
  - Limited form: group = processes to receive a console event

Windows API Thread Creation

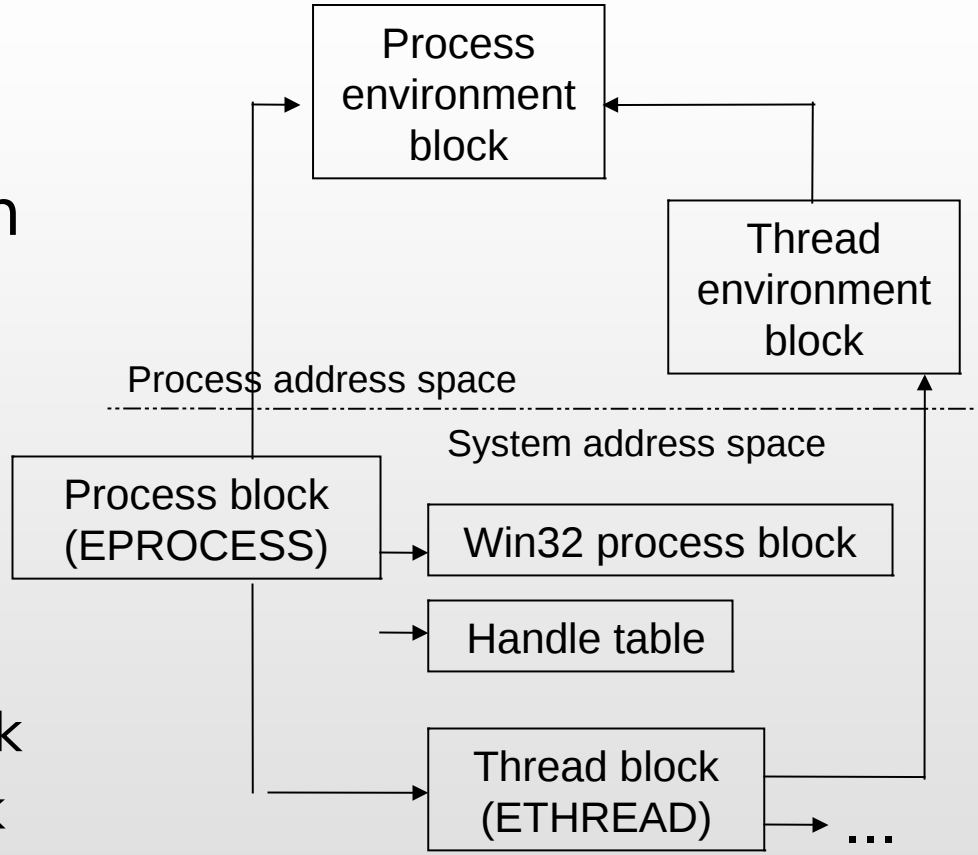
```
HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpsa,
    DWORD cbStack,
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm,
    DWORD fdwCreate,
    LPDWORD lpIDThread)
```

- ☑ lpstartAddr points to function declared as
  DWORD WINAPI ThreadFunc(LPVOID)
- ☑ lpvThreadParm is 32-bit argument
- ☑ LPIDThread points to DWORD that receives thread ID
  non-NULL pointer !

Windows Process and Thread Internals

Data Structures for each process/thread:

- ☑ Executive process block (EPROCESS)
- ☑ Executive thread block (ETHREAD)
- ☑ Win32 process block
- ☑ Process environment block
- ☑ Thread environment block



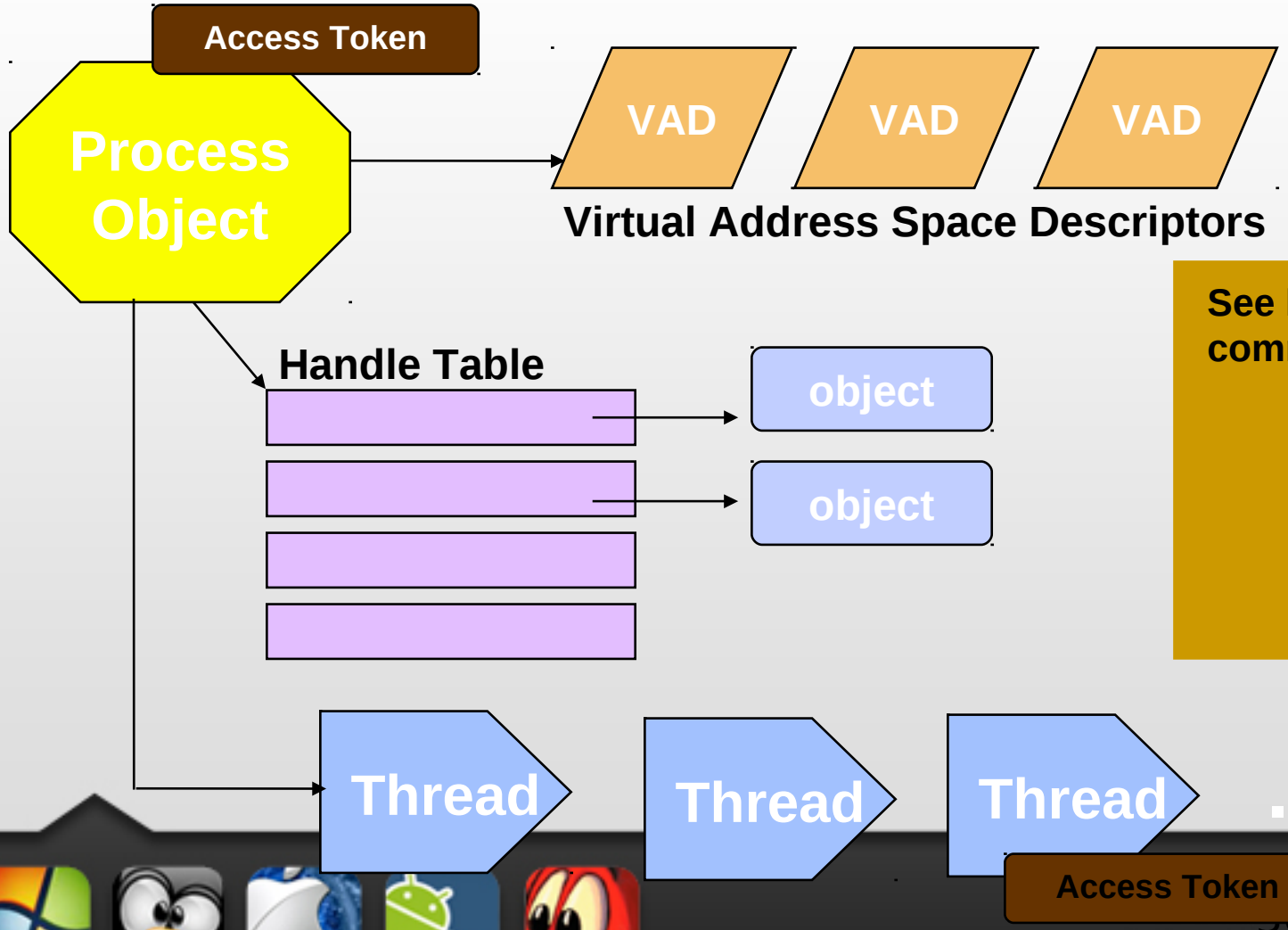
Process

- ☑ Container for an address space and threads
- ☑ Associated User-mode Process Environment Block (PEB)
- ☑ Primary Access Token
- ☑ Quota, Debug port, Handle Table etc
- ☑ Unique process ID
- ☑ Queued to the Job, global process list and Session list
- ☑ MM structures like the WorkingSet, VAD tree, AWE etc

Thread

- ☑ Fundamental schedulable entity in the system
- ☑ Represented by ETHREAD that includes a KTHREAD
- ☑ Queued to the process (both E and K thread)
- ☑ IRP list
- ☑ Impersonation Access Token
- ☑ Unique thread ID
- ☑ Associated User-mode Thread Environment Block (TEB)
- ☑ User-mode stack
- ☑ Kernel-mode stack
- ☑ Processor Control Block (in KTHREAD) for CPU state when not running

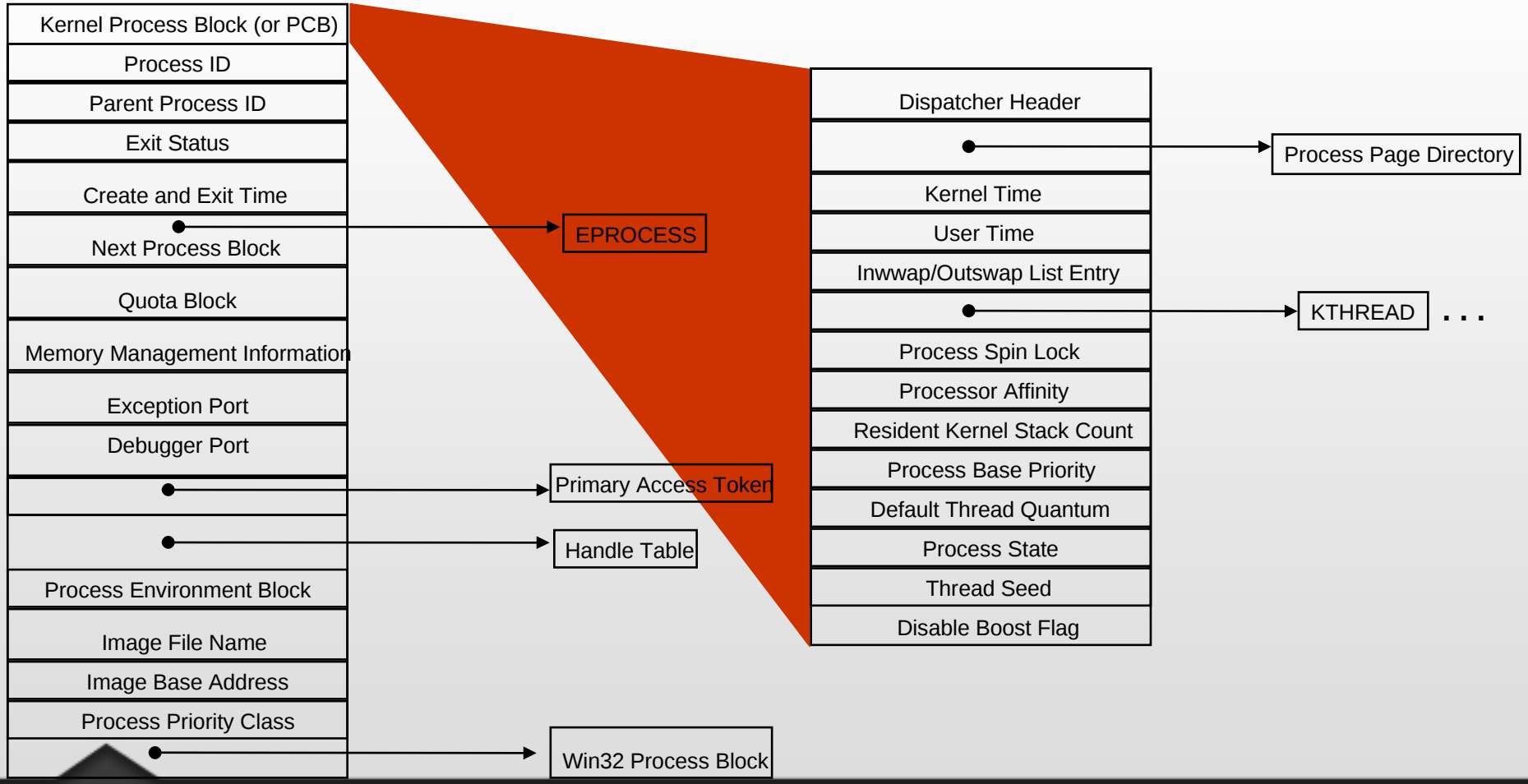
Processes & Threads



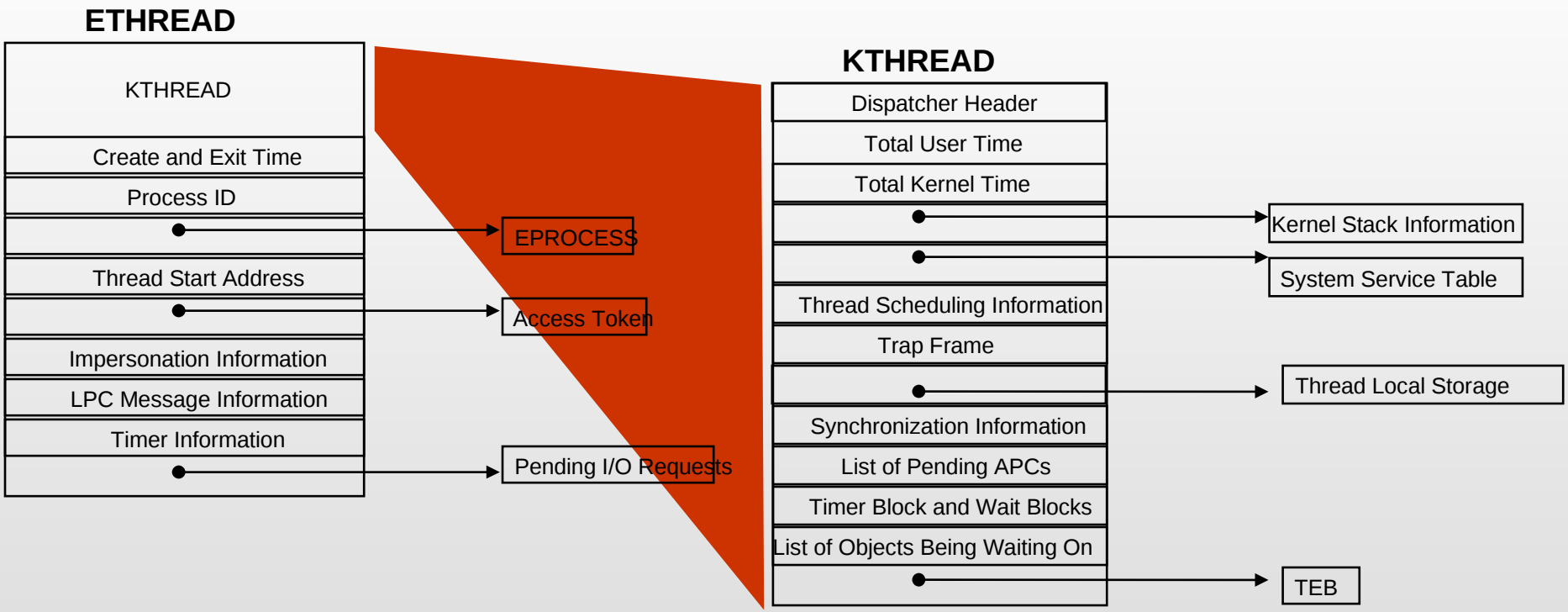
See kernel debugger commands:

- **dt (see next slide)**
- **!process**
- **!thread**
- **!token**
- **!handle**
- **!object**

Process Block

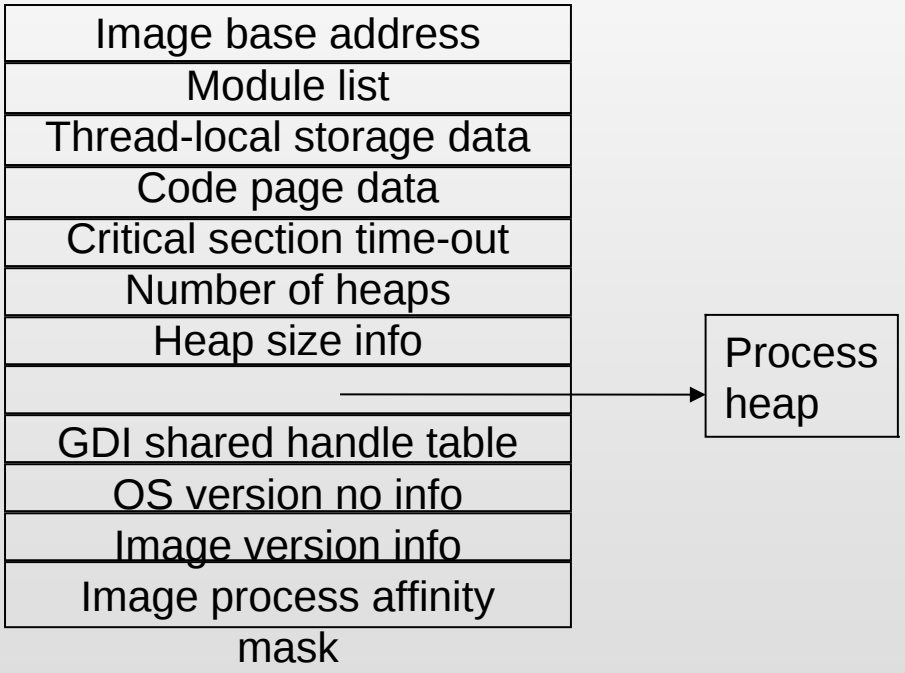


Thread Block



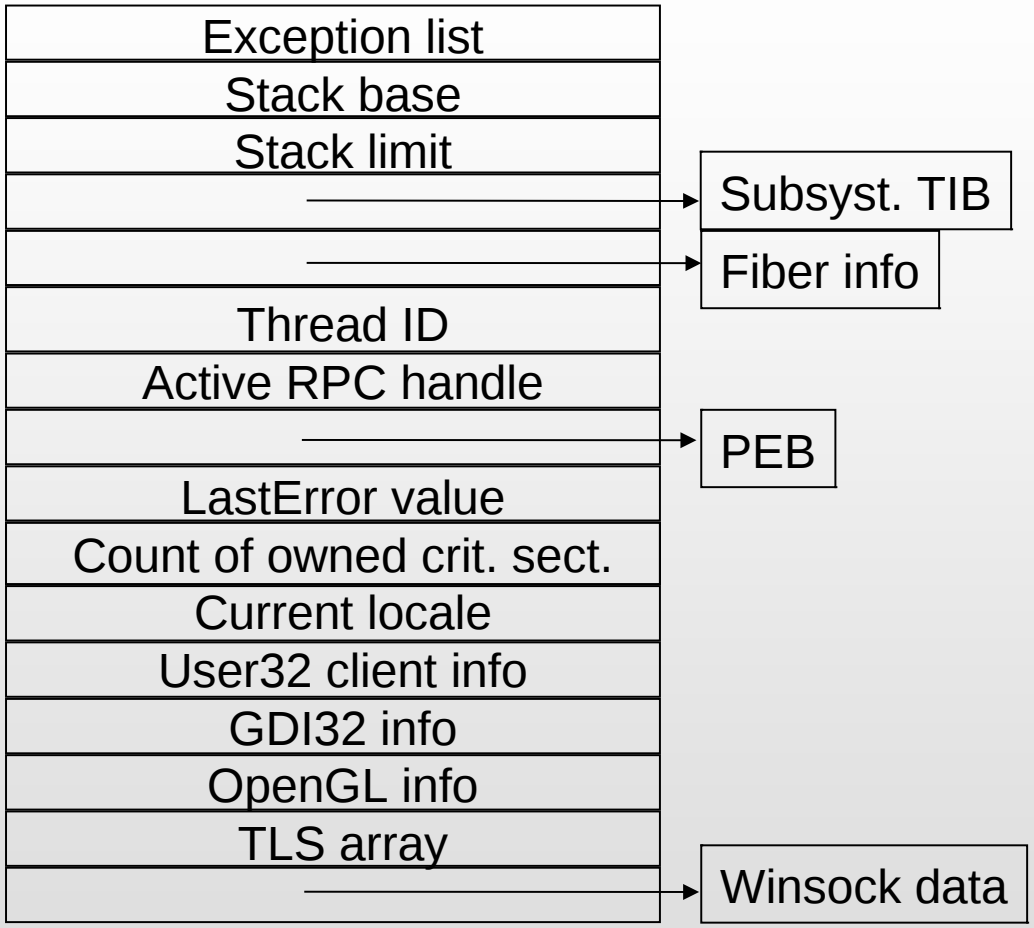
Process Environment Block

- Mapped in user space
- Image loader, heap manager, Windows system DLLs use this info
- View with ! peb or dt nt! _peb



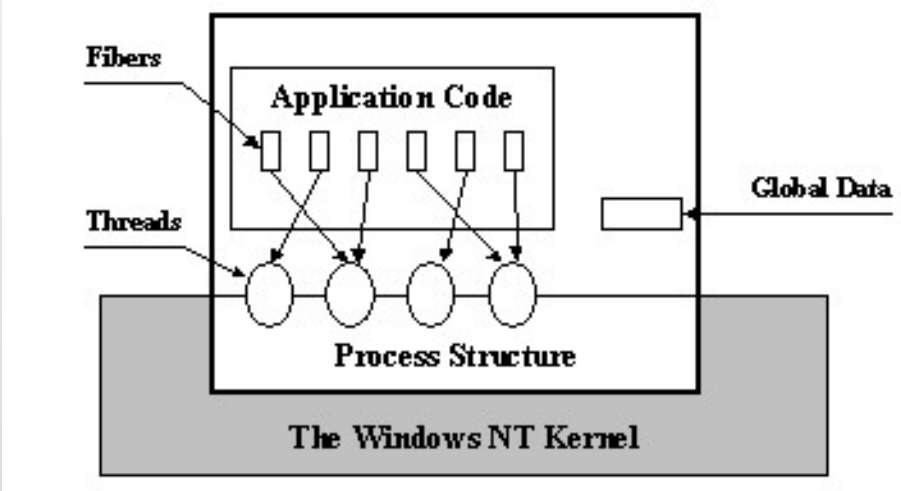
Thread Environment Block

- ☑ User mode data structure
- ☑ Context for image loader and various Windows DLLs
- ☑ View with !teb or dt nt!_teb



Fibras “Fibers”

- ☑ Estructura similar a un thread
- ☑ Ejecuta instrucciones
- ☑ No es mantenida por el Kernel
- ☑ Es creada por el usuario y mantenida y administrada por código de usuario.



Fibras “Fibers” (cont.)

- Corren en el contexto de un thread
- El usuario decide cuando intercambiar entre fibras
- Similar al esquema visto de Solaris
- Cuando el Kernel ejecuta un Thread, este desconoce de la existencia de fibras
- Si un thread es terminado (killed) también todas sus fibras
- Funciones:
  - ConvertThreadToFiber() CreateFiber()
  - GetFiberEnvironment() SwitchToFiber()

Fibras “Fibers” (cont.)

- Tiene la desventaja de no podes ejecutarse en paralelo
  - Solo la fibra activa del thread activo se ejecutara
- Ventaja de poder ser planificadas por el usuario
  - Son creadas en el espacio de usuario
  - No requiere la participación del kernel (menos tiempo de creación)
- Se pueden utilizar en threads que tienen que hacer múltiples tareas, pero solo una en un único momento
- Se pueden utilizar en aplicaciones que necesitan ser portables y deber ser diseñadas para planificar ellas mismas la ejecución de diferentes “hilos”

Fibras “Fibers” (cont.)

- Mas información:
  - http://msdn.microsoft.com/es-ar/library/ms682661(en-us,VS.85).aspx (04/2018)
  - http://msdn.microsoft.com/en-us/library/ms686919(VS.85).aspx (04/2018)

Otras Referencias

- http://www.usenix.org/publications/library/proceedings/usenix-nt98/full_papers/zabatta/zabatta_html/zabatta.html (03/2016)