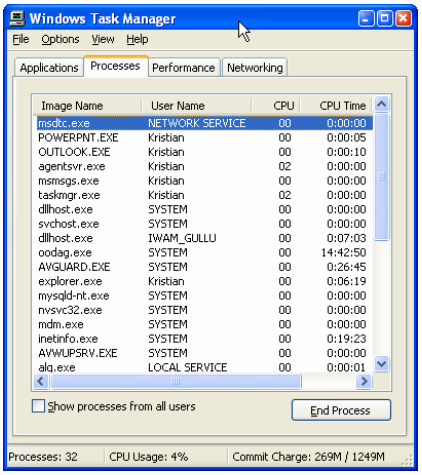
Processes (1)

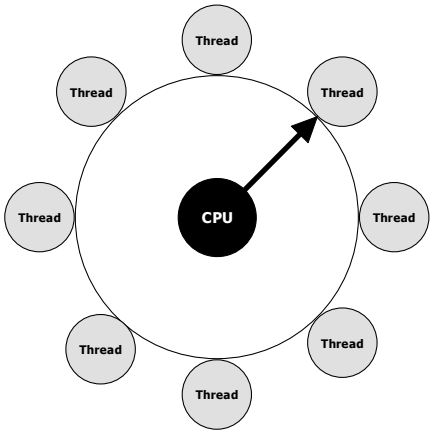
- Sometimes defined as an instance of a running program.
- You can check the processes running on your machine in Task Manager:



Processes (2)

- In Win32 each process owns a 4-GB address space.
- IMPORTANT: a process on its own does not execute anything. For execution a process requires at least on **thread**.
- A process without threads is automatically destroyed.
- When a Win32 process is created, a **Primary Thread** is automatically created for you.
- The primary thread can then create others.

Running Multiple Threads



1. Windows will allocate timeslices (quantums) of CPU time for each thread to execute.
2. Round-Robin scheduling is used (note that threads can have different and changing priorities).
3. This gives the *illusion* that things are executing concurrently.

Environment Variables (1)

- Each process is assigned an **Environment Block**.
- An environment block is simply a portion of allocated memory (owned by the process), containing strings like:

```
VarName1=VarValue1\0
VarName2=VarValue2\0
...
VarNameN=VarValueN\0
\0
```

Environment Variables (2)

- In special cases, environment variables may be used to pass special global parameters to an application.
- In Windows 9X, these variables are set in a special file called autoexec.bat which is parsed when Windows is started. In Windows NT/2000/XP these values can be set from My Computer→Properties.
- Values may be read and written using:
  - `GetEnvironmentVariable`
  - `SetEnvironmentVariable`

Current Drive and Directory

- The **Current Drive** of the **Current Directory** is the default path where Windows looks for files when you try to access them without supplying a fully qualified filename.
- For example if you call the **CreateFile** API call to create a file, if you do not specify the full path, the file will be created in the current directory.
- The current drive/directory is maintained on a per process basis. So all threads in the process will use the same values.
- You can obtain or change the current directories using the following API calls:
  - `GetCurrentDirectory`
  - `SetCurrentDirectory`

Creating Processes (1)

- A process is created when your application is started.
- Note that WinMain isn't really called by the operating system. Instead, it is "expanded" by the compiler.
- Processes are created using the CreateProcess API call (see msdn.microsoft.com for details regarding the function parameters)
- Then the function is called, a process is not actually created. Instead,
  - A small data structure is initialised containing statistical info regarding the process.
  - 4-GB of virtual address space is created.
  - The code and data for the process and associated DLLs are loaded in the address space.

Creating Processes (2)

- Next, a thread is created for the process. This will be the primary thread.
- This thread will eventually run your WinMain function.
- A process can terminate in the following ways:
  - A thread calls the **ExitProcess** API call.
  - A thread in *another process* calles **TerminateProcess** (not very nice).
  - All threads in the process complete.

Process Termination (1)

- A process will terminate when a thread in the process calls ExitProcess.
- ExitProcess is usually automatically called by the primary thread immediately after your WinMain function has completed.
- A separate process can terminate another one by calling TerminateProcess.
- Except in special cases this is discouraged because:
  - When a process terminates properly, all attached DLLs are notified.
  - Using TerminateProcess, this does not happen.

Threads

- Threads must “live” within the context of a process.
- A thread is basically a unit of execution within a process.
- Example: Background printing in Word for Windows.
- On a single processor, threads give the *illusion* that things are happening concurrently.
- Although threads are “cool” and very useful, there are a number of problems associated with them.

Thread Issues (1)

- Consider the following scenario:
  - A user clicks the print button on a Word Processor.
  - The print thread started executing – repaginating, rendering the page, sending to printer, etc…
  - The user can start editing the document when the above is happening.
  - Global Variables:

Thread Issues (2)

```
Global Counter As Long
```

```
Function Calc
  ... {b is 16, x is 1}
  Counter = Sqrt(b)
  Counter = Counter + x*2
  Return Counter
End Function
```

Context Switch

```
Function Calc
  ... {b is 4, x is 3}
  Counter = Sqrt(b)
  Counter = Counter + x*2
  Return Counter
End Function
```

Thread Properties

- Each thread has it's own stack for local variables, etc…
- This stack is allocated from the address space of the main process.
- Static and global variables are shared by all threads in the process.
- Each thread has it's set of CPU registers. A special **Context** structure holds the state of these registers when the thread was last executing.
- This structure is probably the only CPU-specific structure in the API.

Thread Termination

- A thread can terminate in 3 ways:
  - The thread calls the ExitThread API call to terminate itself.
  - Another thread within the same process calls TerminateThread (passing the handle to the thread to terminate) – Webserver monitor thread example.
  - The process "owning" all the threads exits.

Thread Scheduling (1)

- A preemptive operating system must have some defined algorithm for determining when a thread runs and for how long.
- Each thread has a priority ranging from 0 to 31. A thread with priority zero is a special thread used for “memory cleanup”. One system thread has this priority level and no other thread can be assigned this priority.
- The scheduler assigns each priority 31 thread to a CPU to execute.
- Once all priority 31 threads are given a timeslice, another timeslice to each of the priority 31 threads is given.
- This continues until there are no remaining priority 31 threads. Then all priority 30 threads are processed in the same way… and so on.

Thread Scheduling (2)

- Using this technique low priority threads may suffer from a condition known as **Starvation**.
- Also, if a priority 5 thread is running and there is a thread with a higher priority waiting to be serviced, the priority 5 thread is *immediately* suspended for the system to service the higher priority one (even if it is in the middle of a timeslice).

Assigning Priorities (1)

- When a process is created, it is assigned one of 4 priority levels:
  - Idle: Level 4
  - Normal: Level 8
  - High: Level 13
  - Real time: Level 24
- Any thread created in the process will be given that priority as a default.
- The Normal priority level is the one most commonly used.
- The normal priority class is special – it can be “boosted” depending on whether it is a foreground window or not.

Assigning Priorities (2)

- In Windows NT a “boosted” priority is given a bigger timeslice.
- In Windows 9X a “boosted” priority increases the thread priority value by 1 – A “boosted” normal thread has a priority of 8+1=9.
- Real time priority should almost never be used. Even the processes/threads handling the CTRL+ALT+DEL buttons, background disk flushing, mouse and keyboard get a lower priority. This may cause system instability.
- Process “base” priorities can be changed at runtime using the `(Get/Set)PriorityClass` API functions.

Assigning Priorities (3)

- When a thread is created, it is given the priority of the process (base priority).
- You can change the thread's priority relative to the base priority using the `SetThreadPriority` API call.
  - Lowest = Base - 2
  - Below Normal = Base - 1
  - Normal = Base
  - Above Normal = Base + 1
  - Highest = Base + 2
  - Critical = 15, except if the process is real time. Then the priority becomes 31.