**CS490 Windows Internals**

**Solutions for Assignment 3**



Preemptive scheduling:

* Threads are forced (interrupted) by the scheduler to give up the CPU at certain points in time (after quantum expires).
* The Round-Robin scheduling algorithm is preemptive.

Non-preemptive scheduling:

* A running thread may continue execution until it voluntarily gives up CPU.
* The FIFO (FCFS) scheduling algorithm is non-preemptive.

Preemption and context switch is overhead. Therefore, a non-preemptive scheduler would be optimal if maximizing a system’s throughput was the only optimization goal.

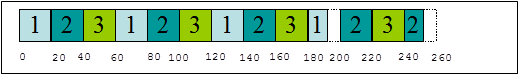
The Windows scheduler in server systems has been optimized towards maximum throughput by using a quantum value six times higher than in client versions of Windows.

Consider a uni-processor system that uses a round-robin algorithm with 16 priorities (0-15, 0 = lowest, 15 = highest priority) for CPU scheduling. The lengths of the time quantum shall be 20ms. Context switching time is negligible. Let us assume that the system schedules threads. Running threads will not be interrupted until quantum end (Scheduling decisions are only be made at quantum end).

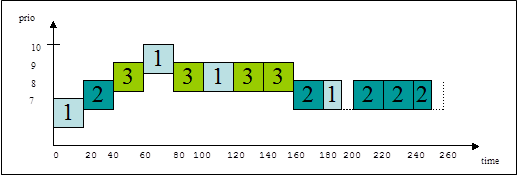
Our system has a workload of three single-threaded processes. These processes (threads) have the following parameters:

|  |  |  |
| --- | --- | --- |
| Thread ID | Ready at | Execution time |
| Th1 | t = 0ms | 70ms |
| Th2 | t = 15ms | 90ms |
| Th3 | t = 30ms | 80ms |

* The Gantt chart below illustrates the execution of these threads under the assumption that they all are executed at a static priority of 8.



* How does the execution order change if Th3 is executed with priority 9. Th1 shall be executed with priority 7 and get into an I/O wait state after 16ms execution time. Th1’s priority is being boosted by 3 after I/O completion.  
  Th1 leaves its wait state at t = 50ms. Th1’s priority then shall be decreased by 1 at quantum end until it reaches its base priority of 7.  
  The corresponding Gantt chart is shown below.



The following C-program implements a version of the UNIX time-command (mytime.exe) using the Windows API. The mytime-command interprets its argument as a program that has to be executed within a separate process and displays the (wall clock) execution time of this process.

/\* Andreas Polze 08-jan-05 \*/

/\* mytime.c \*/

/\* Example solution to assignment 4.4.3 \*/

/\* Windows Operating System Internals \*/

# include <windows.h>

# include <stdio.h>

int main (int argc, char\* argv []) {

STARTUPINFO si;

PROCESS\_INFORMATION pi;

SYSTEMTIME ElTiSys, StartTSys, ExitTSys;

union {

FILETIME ft;

LONGLONG lt;

} CreateT, ExitT, ElapsedT;

if (argc < 1) {

printf("usage: mytime <command>\n");

return -1;

}

GetSystemTime (&StartTSys);

ZeroMemory(&si,sizeof(si));

if (!CreateProcess (NULL, argv[1], NULL, NULL, FALSE,

0, NULL, NULL, &si, &pi)) {

printf("process creation failed, code %i\n",

GetLastError());

return -2;

}

WaitForSingleObject (pi.hProcess, INFINITE);

GetSystemTime (&ExitTSys);

SystemTimeToFileTime (&StartTSys, &CreateT.ft);

SystemTimeToFileTime (&ExitTSys, &ExitT.ft);

/\* ---- 64bit arithmetic on FILETIME ---- \*/

ElapsedT.lt = ExitT.lt - CreateT.lt;

FileTimeToSystemTime (&ElapsedT.ft, &ElTiSys);

printf ("Real Time: %02d:%02d:%02d:%03d\n",

ElTiSys.wHour, ElTiSys.wMinute, ElTiSys.wSecond,

ElTiSys.wMilliseconds);

CloseHandle (pi.hThread);

CloseHandle (pi.hProcess);

return 0;

}