**QNX: A guide for Realtime programmers**

**Analogy for processes and threads**

1. A process as a house: A house is really a container, with certain attributes (such as the amount of floor space, the number of bedrooms, and so on). If you look at it that way, the house really doesn't actively do anything on its own — it's a passive object.
2. The occupants as threads: The people living in the house are the active objects — they're the ones using the various rooms, watching TV, cooking, taking showers, and so on.
   1. Single threaded: If you've ever lived on your own, you know that you can do anything you want in the house at any time, because there's nobody else in the house.
   2. Multi threaded: Things change dramatically when you add another person into the house. Let's say you get married, so now you have a spouse living there too. You can't just march into the washroom at any given point; you need to check first to make sure your spouse isn't in there. If you have two responsible adults living in a house, generally you can be reasonably lax about “security” — you know that the other adult will respect your space, won't try to set the kitchen on fire (deliberately!), and so on.

Now, throw a few kids into the mix and suddenly things get a lot more interesting.

**1. Process and threads**

A process occupies memory and threads have common access to that memory(hence, access must be synchronized across threads, or memory must be declared private for a particular thread so no synchronization is required). However, threads depend on the number of processors, because a processor can only execute one thread at any time.

**Mutual exclusion**

a number of threads are mutually exclusive when it comes to a shared resource, this is communicated via an object called MUTEX which basically locks the resource for exclusive uses and the threads decides when to release(unlock) it.

**Priorities**

who gets an unlocked resource first is done via two factors: primarily by priority(1- N) and secondary by length of wait(time).

**Semaphores**

limits the number of threads that have access to a resource by counting and stablishing a maximum value, a Mutex is the same as a semaphore with count of 1. However the MUTEX is special purpose and therefore most efficient.

**2. The Kernel’s role**

**Single CPU**

In this case, since there's only one CPU present, only one thread can run at any given point in time. The kernel decides (using a number of rules, which we'll see shortly) which thread to run, and runs it.

**Multiple CPU (SMP)**

If you buy a system that has multiple, identical CPUs all sharing memory and devices, you have an SMP box (SMP stands for Symmetrical Multi Processor, with the “symmetrical” part indicating that all the CPUs in the system are identical). In this case, the number of threads that can run concurrently (simultaneously) is limited by the number of CPUs. Since each processor can execute only one thread at a time, with multiple processors, multiple threads can execute simultaneously

**The kernel as arbiter**

The kernel determines which thread should be using the CPU at a particular moment, and switches context(save current thread registers and other context information and load new thread’s registers and context into CPU) to that thread. The kernel checks the state of its threads and if many threads can consume CPU, then uses a priority(1-N) and scheduling policy(FIFO or round robin) evaluated in that order.

Scheduling Policies:

- FIFO: Threads consumes CPU for as long as it wants. If the thread quits or voluntarily gives up the CPU, then the kernel looks for other threads at the same priority or afterwards for lower-priority threads. Yield gives up CPU ONLY to another thread in the same priority.

- Round Rovin: identical to FIFO, except that the thread will not run forever if there is anothre thread at the same priority. RR divides time into system-defined time slices and if there are 2 threads running on the same priority, then CPU switches between these 2.

**kernel states**

running: actively consuming CPU.

Ready: could run but another thread is running.

Blocked: keeps track of “why” the thread is blocked(due to waiting for: variable, dead(release resources), interrupt, completion of another thread, mutex, sleep, receive, stopped and more)

**3. Threads and processes as a system**

Each process is responsible for providing a service of some nature(file-system, display driver, data acquisition, control module…).The advantages of breaking things up into multiple processes(instead of 1 process and a zillion threads) are:

- decoupling and modularity

- maintainability

- reliability

A processor is in isolation from other processors, so each module is secure and independent(hence easy to maintain). The only reliance with other processes is through some well-defined interfaces. This reliability is the most important point, a process has some well-defined “borders”, if a thread tries to access memory out of the process it gets killed so two threads running on different processes are isolated from each other.

**Memory protection**: process address space allows efficient context-switch operation between threads in the same process. A little overhead is added if the threads are on different process, as it requires address space switch as well.

**Starting a process**: Unix function calls for threads and processes

$ program1 → start a program1 and wait to finish

$ nice program2 → start a program2 at reduced priority(adjusting its own priority to low, and child threads inherit the priority).

System() → takes a command line and executes it, just as if you type it at a shell prompt. eg. system(“pwd”)

exec() → transform current process into another

spawn() →

fork() →

vfork() →