SNS COLLEGE OF TECHNOLOGY

Coimbatore-35

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

23CST202- OPERATING SYSTEMS

II YEAR AIML B IV SEM

UNIT 1 – OVERVIEW AND PROCESS MANAGEMENT

TOPIC – THREADS –MULTI THREADING MODELS

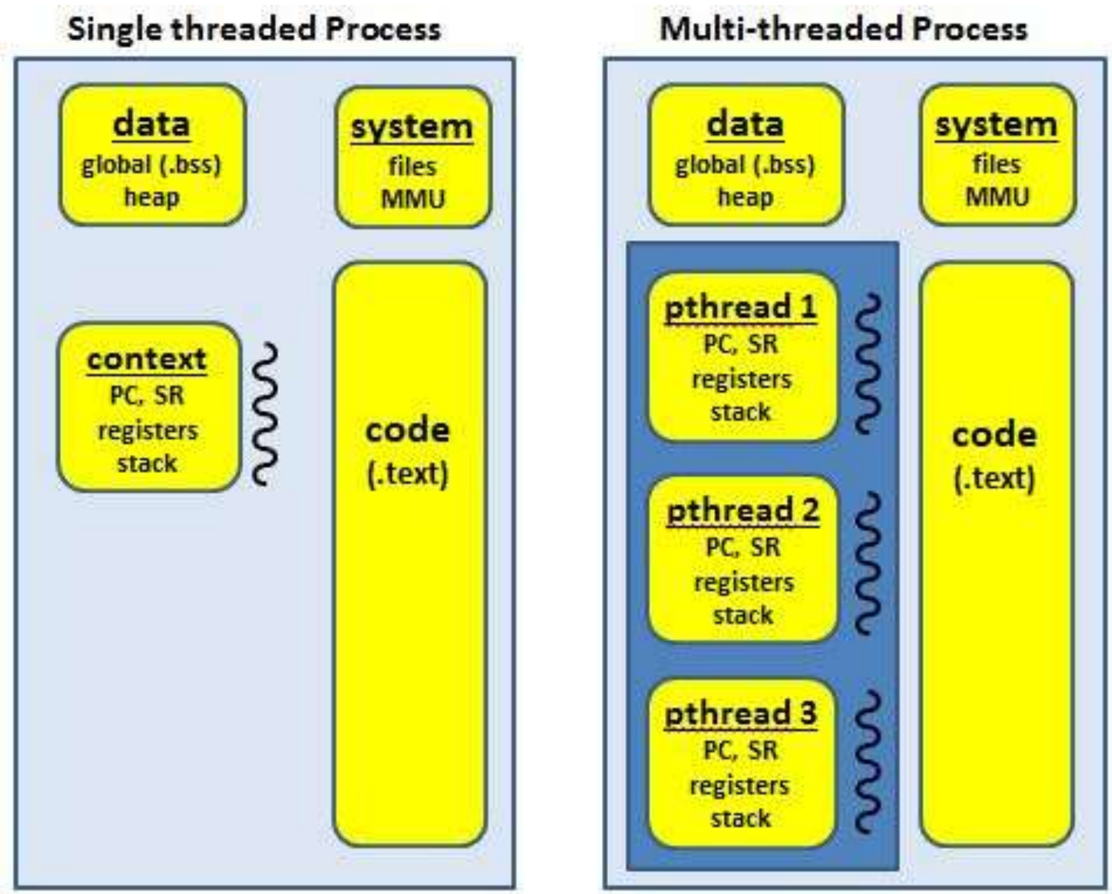
Threads (Lightweight)

- What is a thread?
  - An independent program counter and stack operating within a process - sometimes called a lightweight process (LWP)
  - Smallest unit of processing (context) that can be scheduled by an operating system
- What resources are owned by a thread?
  - CPU registers (PC, SR, SP, ...)
  - Stack
  - State
- What do all process threads have in common?
  - Process resources
  - Global variables
- How would you describe inter-thread communication?
  - Cheap: can use process memory without needing a context switch.
  - Not Secure: one thread can write to memory in use by another thread.

Types of Threads

- A thread consists of:
  - a thread execution state (Running, Ready, etc.)
  - a context (program counter, register set.)
  - an execution stack.
  - some per-tread static storage for local variables.
  - access to the memory and resources of its process (shared with all other threads in that process.)
  - OS resources (open files, signals, etc.)
- Thus, all of the threads of a process share the state and resources of the parent process (memory space and code section.)
- There are two types of threads:
  - User-space (ULT) and
  - Kernel-space (KLT).

Multi-threading



Task Control Block (tcb)

State = { NEW, READY, RUNNING, BLOCKED, EXIT
Priority = { LOW, MED, HIGH, VERY_HIGH, HIGHEST }

```
// task control b
typedef struct
{
     char* name;                         // task name
     int (*task)(i          r**);        // task address
     int state;                          // task state (P2)
     int priority;                       // task priority (P2)
     int argc;                           // task argument count (P1)
     char** argv;                        // task argument pointers (P1)
     int signal;
                                           // task signals (P1)
//   void (*sigContHandler)(void);       // task mySIGCONT handler
     void (*sigIntHandler)(void);        // task mySIGINT handler
//   void (*sigKillHandler)(void);
//   void (*sigTermHandler)(void);
//   void (*sigTstpHandler)(void);       //      mySIGTSTP handler
     TID parent;                         //  task parent
     int RPT;                            // task root page table (P4)
     int cdir;                           // task directory (P6)
     Semaphore *event;                   // blocked task semaphore (P2)
     void* stack;                        // task stack (P1)
     jmp_buf context;                    // task context pointer (P1)
} TCB;
```

Pending semaphore when blocked.

User-Level Threads

- User-level threads avoid the kernel and are managed by the process.
  - Often this is called "cooperative multitasking" where the task defines a set of routines that get "switched to" by manipulating the stack pointer.
  - Typically each thread "gives-up" the CPU by calling an explicit switch, sending a signal or doing an operation that involves the switcher.
  - A timer signal can force switching.
  - User threads typically can switch faster than kernel threads [however, Linux kernel threads' switching is actually pretty close in performance].

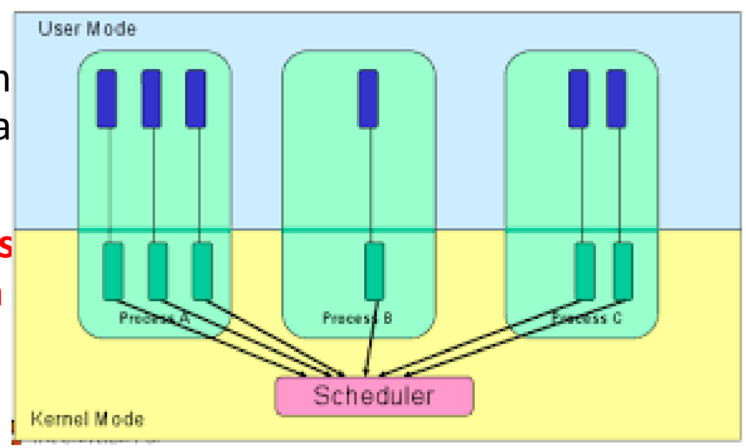
User-Level Threads

- Disadvantages.
  - User-space threads have a problem that a single thread can monopolize the timeslice thus starving the other threads within the task.
  - Also, it has no way of taking advantage of SMPs (Symmetric MultiProcessor systems, e.g. dual-/quad-Pentiums).
  - Lastly, when a thread becomes I/O blocked, all other threads within the task lose the timeslice as well.
- Solutions/work arounds.
  - Timeslice monopolization can be controlled with an external monitor that uses its own clock tick.
  - Some SMPs can support user-space multithreading by firing up tasks on specified CPUs then starting the threads from there [this form of SMP threading seems tenuous, at best].
  - Some libraries solve the I/O blocking problem with special wrappers over system calls, or the task can be written for nonblocking I/O.

Kernel-Level Threads

- KLTs often are implemented in the kernel using several tables (each task gets a table of threads).
  - The kernel schedules each thread within the timeslice of each process.
  - There is a little more overhead with mode switching from user to kernel mode because of loading of larger contexts, but initial performance measures indicate a negligible increase in time.

- Advantages.
  - Since the clocktick will determ … ly to hog the timeslice from the other threa
  - I/O blocking is not a problem.
  - **If properly coded, the process … /IPs and will run incrementally faster with**



User-Level and Kernel-Level Threads