Introduction to OS

Threads

MOS 2.2

Mahmoud El-Gayyar

elgayyar@ci.suez.edu.eg

Problem

- Processes in Unix, Linux, and Windows are “heavyweight”
- OS-centric view – performance
- Application-centric view – flexibility and application design

OS-centric View of Problem

- Lots of data in PCB & other data structures
  - Even more when we study memory management
  - More than that when we study file systems, etc.
- Processor caches a lot of information
  - Memory Management information
  - Caches of active pages
- Costly context switches and traps
  - 100’s of microseconds

Application-centric View of Problem

- Separate processes have separate address spaces
  - Shared memory is limited or nonexistent
  - Applications with internal concurrency are difficult
- Isolation between independent processes *vs.* cooperating activities
  - Fundamentally different goals

Example

- Web Server – How to support multiple concurrent requests
- One solution:
  - create several processes that execute in parallel
- inefficient
  - *Space and time:* PCB, page tables, cloning entire process, etc.

Example 2

- Transaction processing systems
  - E.g, airline reservations or bank ATM transactions
- 1000's of transactions *per second*
  - Very small computation per transaction
- Separate processes per transaction are too costly

Example 3

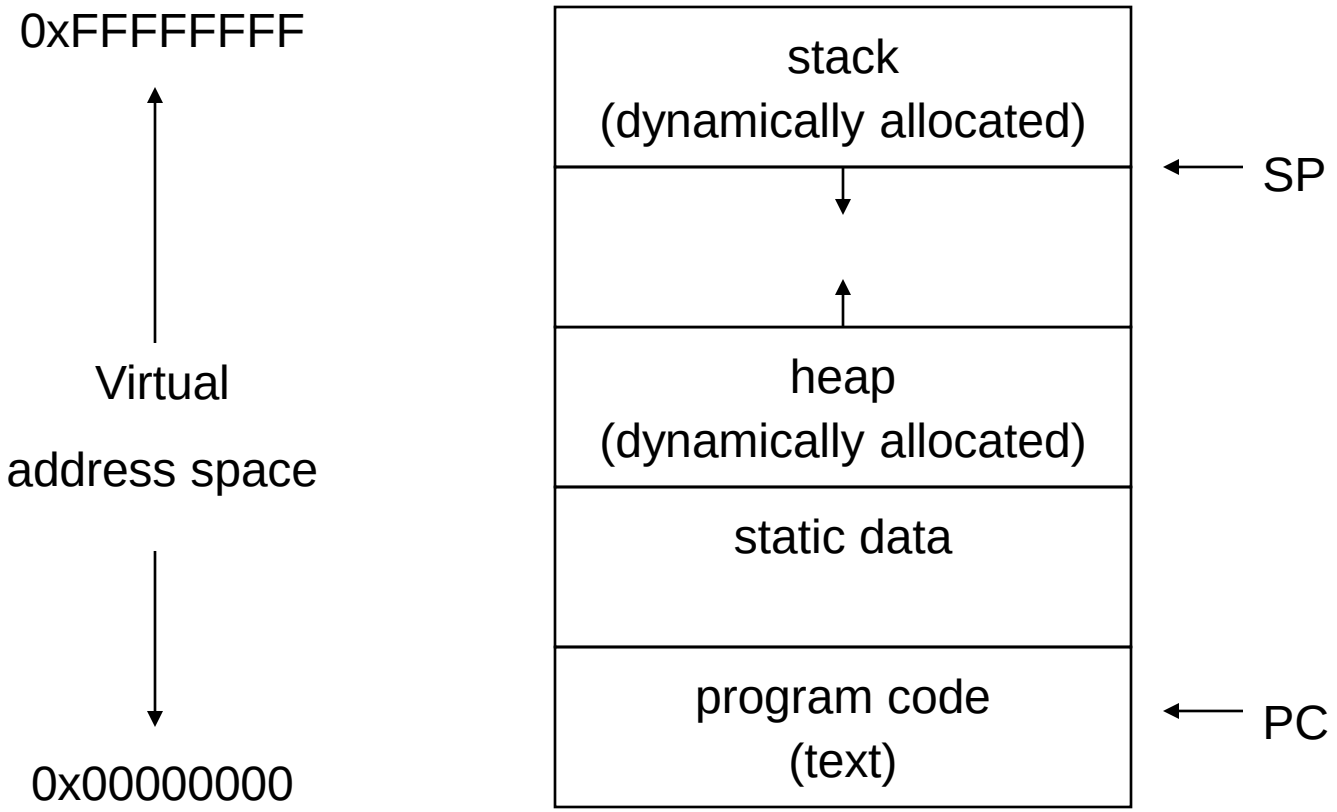
- Games have multiple active characters
  - Independent behaviors
  - Common context or environment
- Need “real-time” response to user
  - For interactive gaming experience
- Programming all characters in separate processes is really, really hard!
- Programming them in a single process is much harder without concurrency support.

Solution:– *Threads*

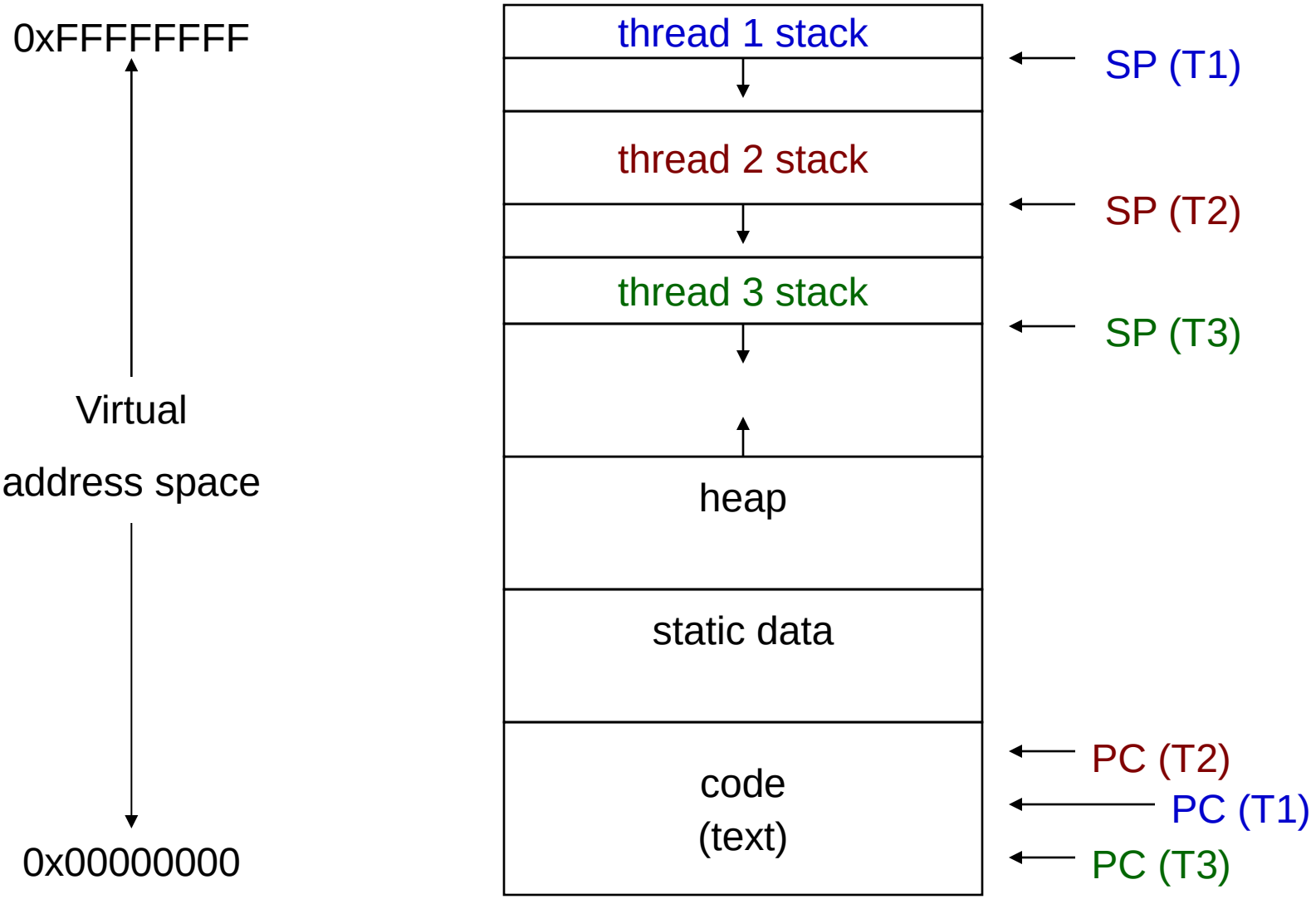
- A *thread* is a particular execution of a program, function, or procedure *within the context* of a Unix or Windows process
  - I.e., a specialization of the concept of *process*
- A thread ***has its own***
  - Program counter, registers, PSW
  - Stack
- A thread ***shares***
  - Address space, heap, static data, program code
  - Files, privileges, all other resources

  with all other threads of the same process

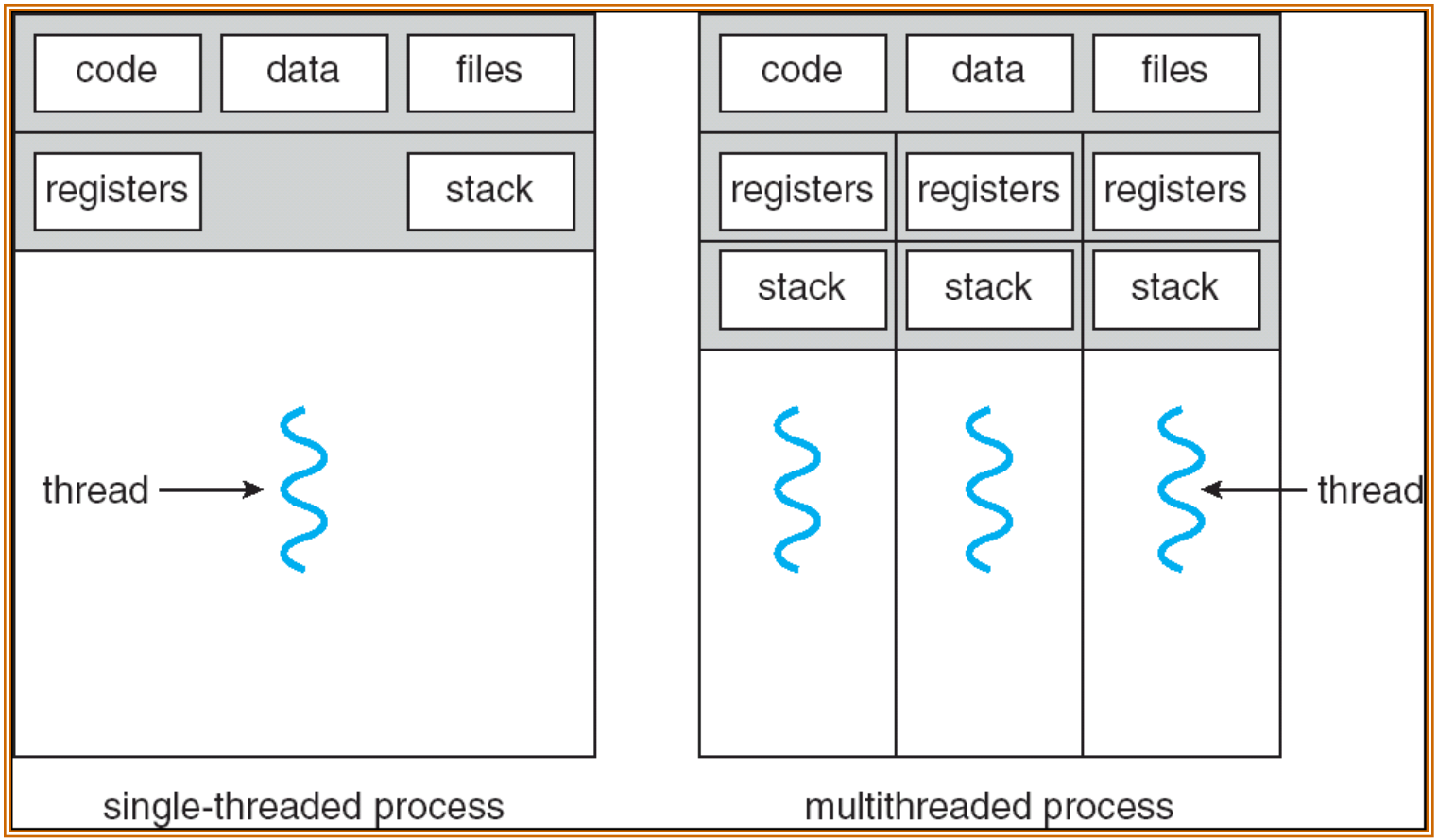
Address Space
Linux-Windows process



Address Space for Multiple Threads



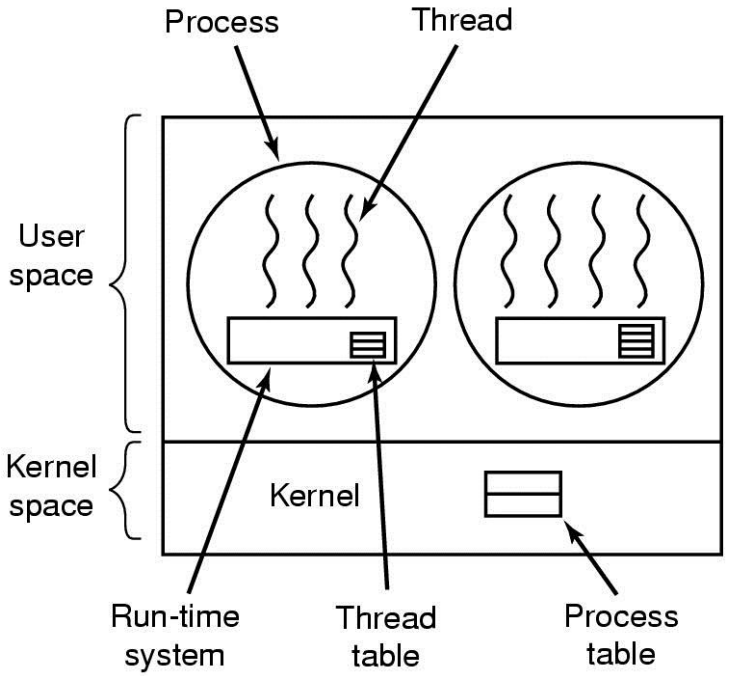
Single and Multithreaded Processes

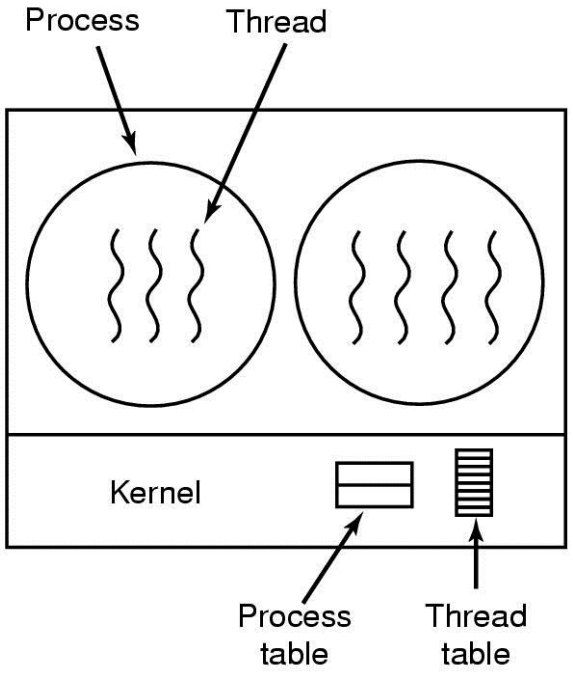


Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of multi-processor architectures

User vs. Kernel Threads





Implementation of Threads

- ***User*-level implementation**
  - User-space function library
  - Runtime system – similar to process management except in user space
  - Windows NT – *fibers:* a user-level thread mechanism

- ***Kernel* implementation – primitive objects known to and scheduled by kernel**
  - Linux: *lightweight process* (LWP)
  - Windows NT & XP: *threads*

User Threads

- Can be implemented without kernel support
  - … or knowledge!
- Program links with a runtime system that does thread management
  - Operation are very efficient (procedure calls)
  - Space efficient and all in user space (TCB)
  - Task switching is very fast
- Since kernel not aware of threads, there can be scheduling inefficiencies
  - E.g., blocking I/O calls
  - Non-concurrency of threads on multiple processors

User Threads (continued)

- Thread Dispatcher
  - Queues in process memory to keep track of threads' state
- Scheduler – non-preemptive
  - Assume threads are **well-behaved**
  - Thread voluntarily gives up CPU by calling *yield*() – does thread context switch to another thread
- Scheduler – preemptive
  - Assumes threads may not be well-behaved
  - Scheduler sets timer to create a *signal* that invokes scheduler
  - Scheduler can force thread context switch
  - Increased overhead
- Application or thread library must handle *all* concurrency itself!

Kernel Threads

- Supported by the Kernel
  - OS maintains data structures for thread state and does all of the work of thread implementation.
- Examples
  - Windows XP/2000
  - Solaris
  - Linux version 2.6
  - Tru64 UNIX
  - Mac OS X

Kernel Threads (continued)

- OS schedules *threads* instead of *processes*
- Benefits
  - Overlap I/O and computing in a process
  - Creation is cheaper than processes
  - Context switch *can* be faster than processes
- Negatives
  - System calls (high overhead) for operations
  - Additional OS data space for each thread

Thread Libraries

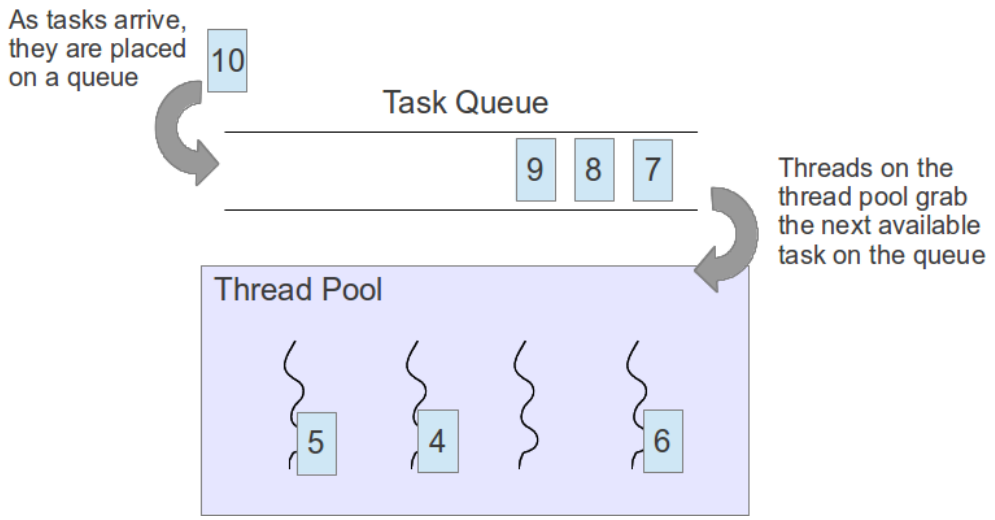
- Thread management done by a thread library
- Three primary thread libraries: –
  - ***POSIX Pthreads*** - may be provided as either a user or kernel library
  - ***Win32 threads*** - provided as a kernel-level library on Windows systems.
  - ***Java threads*** - Java runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

Unix Processes *vs.* Threads

- On a 700 Mhz Pentium running Linux
  - Processes:
    - **`fork()`**: 250 microsec
  - Kernel threads:
    - **`pthread_create()`**: 90 microsec
  - User-level threads:
    - **`pthread_create()`**: 5 microsec

Thread Pools (Implementation technique)

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool



Threads – Summary

- Threads were invented to counteract the heavyweight nature of *Processes* in Unix, Windows, etc.
- Provide lightweight concurrency *within* a single address space
- Have evolved to become *the* primitive abstraction defined by kernel
  - Fundamental unit of scheduling in Linux, Windows, etc

Review

- Shall a thread have its own registers?
- Can a thread ever be preempted by a clock interrupt? If so, under what circumstances? If not, why not?
- What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?