Operating Systems

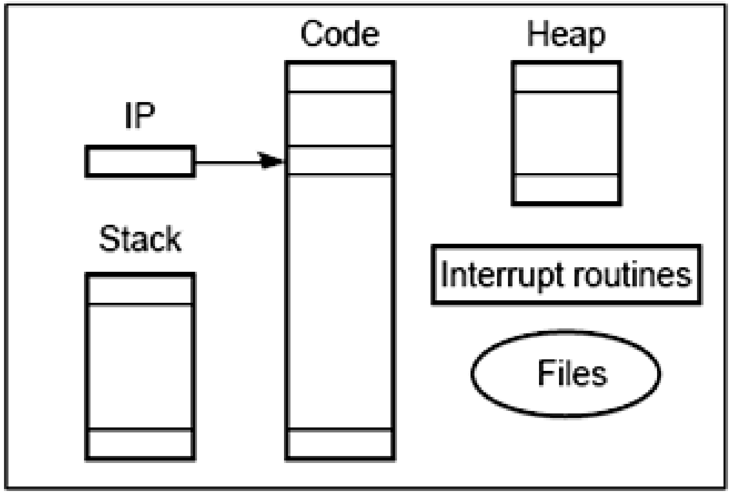
Concepts of Threads

- Concept of Process has two facets.
- A Process is:
  - ✓A Unit of resource ownership:
    - ❑a virtual address space for the process image
    - ❑control of some resources (files, I/O devices...)
  - ✓A Unit of execution - process is an execution path through one or more programs
    - ❑may be interleaved with other processes
    - ❑execution state (Ready, Running, Blocked...) and dispatching priority

- These two characteristics are treated separately by some recent operating systems:
  - ✓The *unit of resource ownership* is usually referred to as a *process* or *task*
  - ✓The *unit of execution* is usually referred to a *thread* or a “lightweight process”

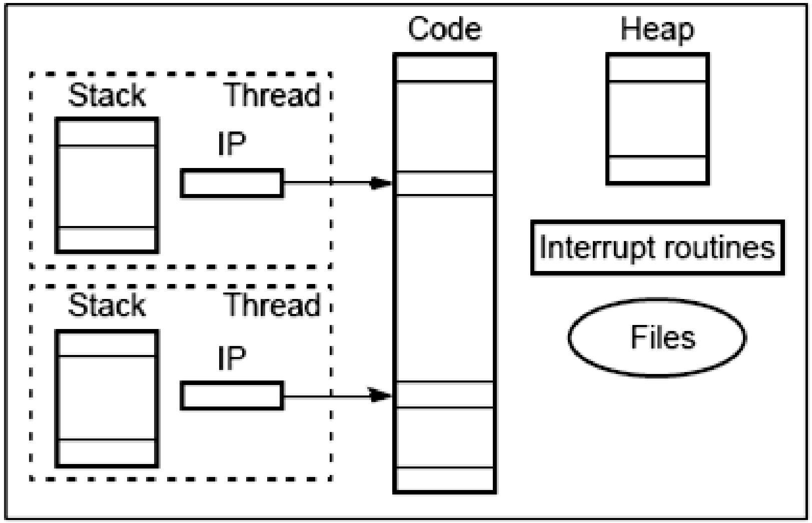
- **Multithreading:** The OS supports multiple threads of execution within a single process
- **Single threading:** The OS does not recognize the separate concept of thread
  - ✓MS-DOS supports a single user process and a single thread
  - ✓Traditional UNIX supports multiple user processes but only one thread per process
  - ✓Solaris and Windows 2000 support multiple threads

Processes



- “Heavyweight” process
- Completely separate program with its own
  - ✓ Process
  - ✓ Variables
  - ✓ Stack
  - ✓ Memory allocation

Threads



- Lightweight process
- Shares the same memory space and global variables between routines.

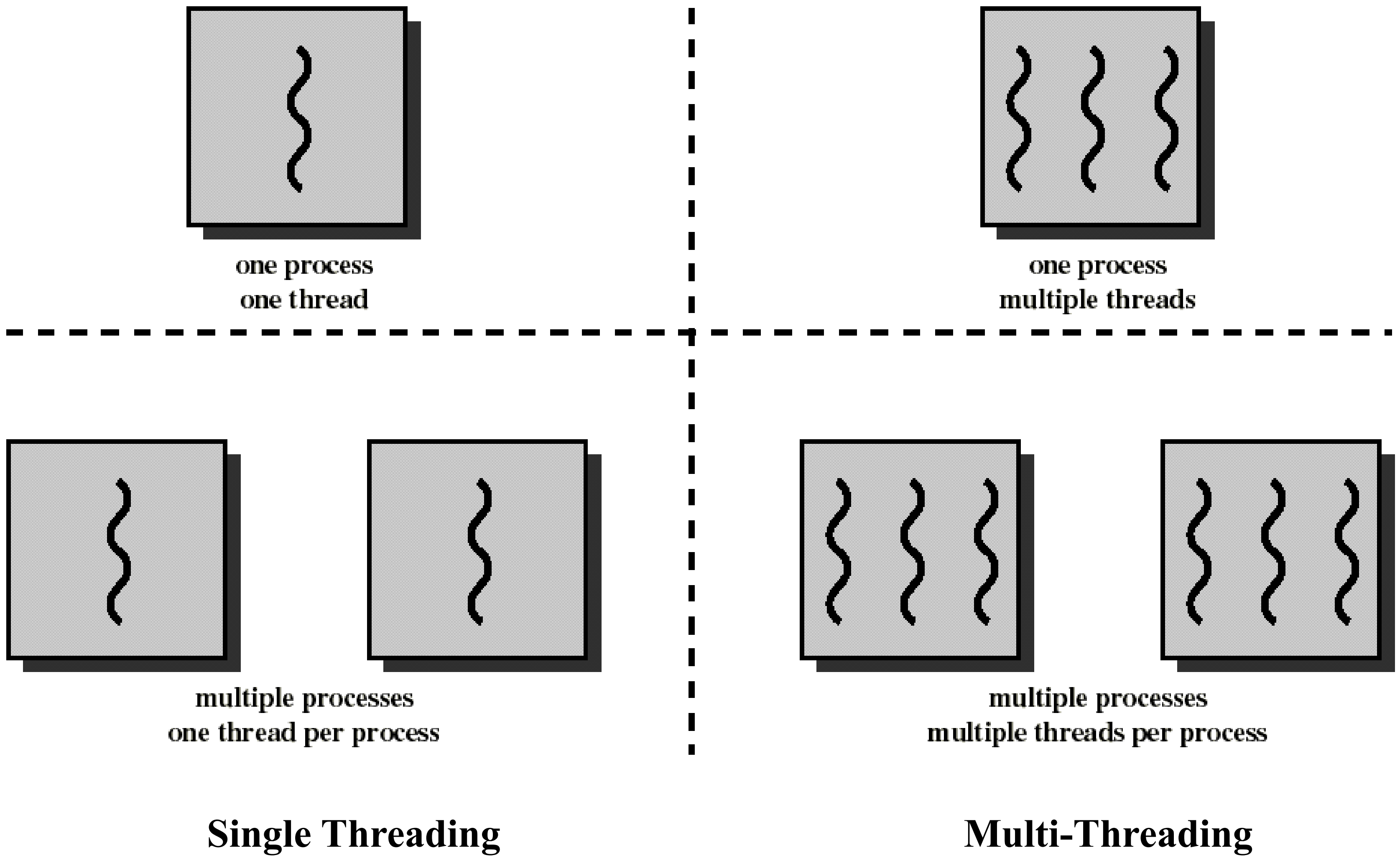
- Processes Have:
  - ✓ A virtual address space which holds the process image
  - ✓ Protected access to processors, other processes (inter-process communication), files, and other I/O resources
- Threads Have:
  - ✓ Have execution state (running, ready, etc.)
  - ✓ Save thread context (e.g. program counter) when not running
  - ✓ Have private storage for **local** variables and **execution stack**
  - ✓ Have **shared access** to the address space and resources (files etc.) of their process
    - ❑ when one thread alters (non-private) data, all other threads (of the process) can see this
    - ❑ threads communicate via shared variables
    - ❑ a file opened by one thread is available to others

Threads

- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls main & has the process*'s stack*
- Inexpensive creation
- Inexpensive context switching
- Efficient communication
- If a thread dies, its stack is reclaimed

Processes

- A process has code/data/heap & other segments
- A process has at least one thread
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- Interprocess communication can be expressive
- If a process dies, its resources are reclaimed & all threads die


one process
one thread
one process
multiple threads
multiple processes
one thread per process
multiple processes
multiple threads per process
Single Threading
Multi-Threading

- **Responsiveness** - Opens the possibility of blocking only one thread in a process on a blocking call rather than the entire process.
- **Resource Sharing** – Threads share resources of process to which they belong – code sharing allow multiple instantiations of code execution
- **Economy** – low overhead in thread context switching & thread management compared to process context switching & management
- **Utilization of MP Architectures** – can be applied to a multi-processor. In a uniprocessor, task switching is so fast that it gives illusion of parallelism.

- Consider an application that consists of several independent parts that do not need to run in sequence
- Each part can be implemented as a thread
- Whenever one thread is blocked waiting for I/O, execution could switch to another thread of the same application (instead of switching to another process)
- Example 1: File Server on a LAN
  - ✓ Needs to handle many file requests over a short period
  - ✓ Threads can be created (and later destroyed) for each request
  - ✓ If multiple processors: different threads could execute simultaneously on different processors
- Example 2: Spreadsheet on a single processor machine:
  - ✓ One thread displays menu and reads user input while the other executes the commands and updates display

- Three key states: Running, Ready, Blocked
- No Suspend state because all threads within the same process share the same address space (same process image)
  - ✓ Suspending implies swapping out the whole process, suspending all threads in the process
- Termination of a process **terminates all threads** within the process
  - ✓ Because the process is the environment the thread runs in

Spawn:

- Process starts with one thread. That thread can spawn another thread, placing the new thread on the Ready queue

Block (yield, suspend):

- Save PC, registers, etc. and allow other thread(s) to run
- Could “block” whole process if making system call which requires kernel service, otherwise it’s a single thread being suspended.

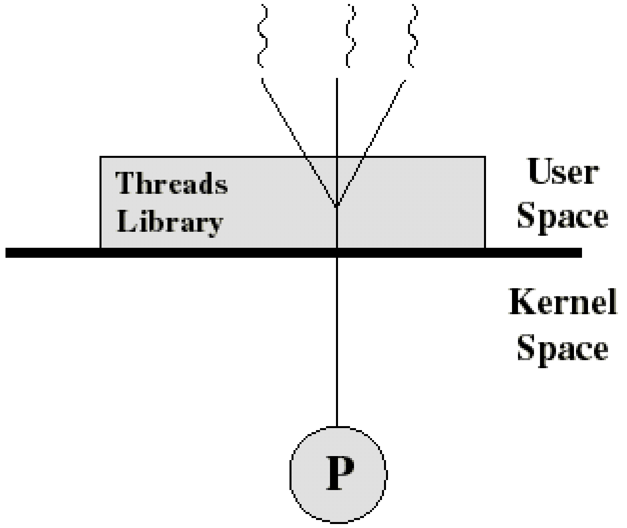
Unblock (wake):

- IO finishes, or another relinquishes control, thread moves to Ready queue

Finish (terminate):

- Deallocate context (stacks etc.)

- Kernel not aware of the existence of threads
- Thread management handled by thread library in user space
- No mode switch (kernel not involved)
- But I/O in one thread could block the entire process!
- application concurrency, how many tasks?
- Examples
  - ✓ POSIX *Pthreads*
  - ✓ Mach *C-threads*
  - ✓ Solaris *threads*



“Many-to-One” model

- Contains code for:
  - ✓ creating and destroying threads
  - ✓ passing messages and data between threads
  - ✓ scheduling thread execution
    - ❑ pass control from one thread to another
  - ✓ saving and restoring thread contexts

- ULT's can be be implemented on **any** Operating System, because **no kernel services are required** to support them

- The kernel is **not aware** of thread activity
  - ✓ it only manages processes
- If a thread makes an I/O call, **the whole process** is blocked
  - ✓ Note: in the thread library that thread is still in “**running**” state, and will resume execution when the I/O is complete
- So **thread states** are independent of **process states**

Advantages

- Fast: Thread switching does not involve the kernel

  no system calls required
- Portable: few system dependencies

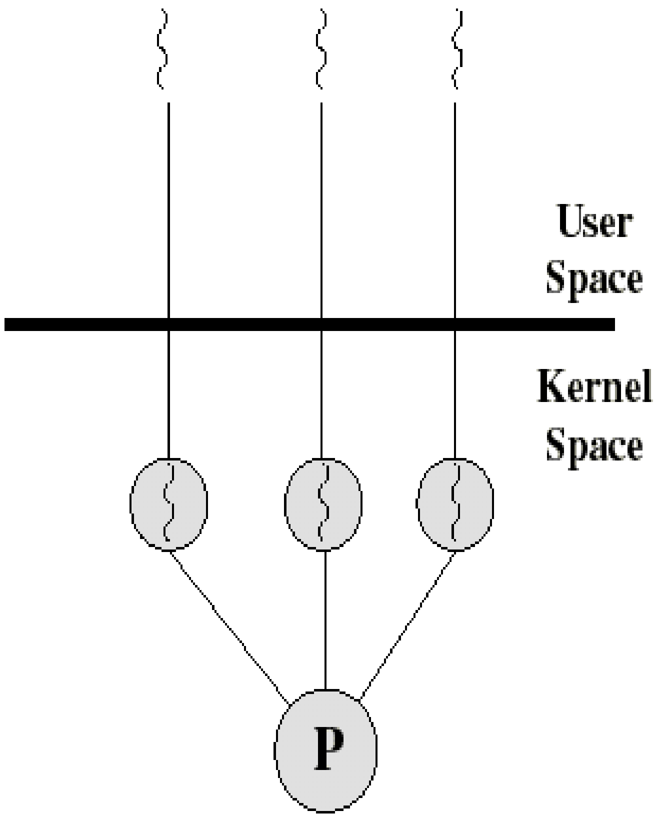
  Can run on any OS. We only need a thread library
- Scheduling can be application specific: choose the best algorithm for the situation.

Disadvantages

- Most system calls are blocking for processes. So all threads within a process will be implicitly blocked
- No parallel execution of threads
- The kernel can only assign processors to processes. Two threads within the same process cannot run simultaneously on two processors

Ex: Windows NT, Windows 2000, OS/2

- All thread management is done by kernel
- No thread library; instead an API to the kernel thread facility
- Kernel maintains context information for the process and the threads
- Switching between threads requires the kernel
- Kernel does Scheduling on a thread basis
- physical concurrency, how many cores?



"One-to-One" model

Advantages

- **More concurrency:** The kernel can schedule multiple threads of the same process on multiple processors
- Blocking at thread level, not process level
  - ✓ If a thread blocks, the CPU can be assigned to another thread in the same process
- Even the kernel routines can be multithreaded

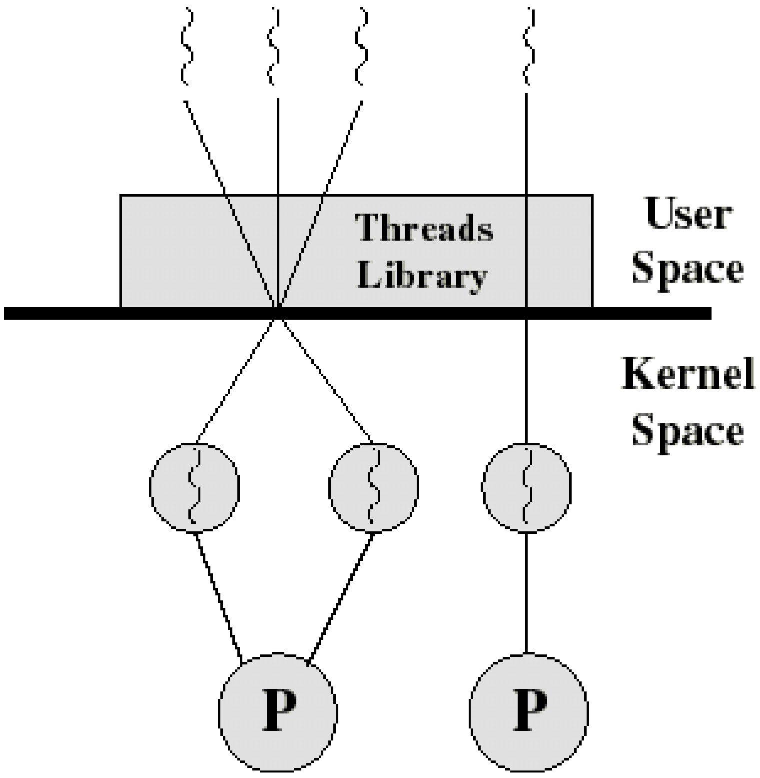
Disadvantages

- **Expensive:** Thread switching **always** involves the kernel. This means 2 mode switches per thread switch
- Thread need kernel resources
- So it is **slower** compared to User Level Threads
  - ✓(But **faster** than a full process switch)

(e.g. Solaris)

- Many user-level threads map to many kernel threads (U >= K)

  The programmer may adjust the number of KLTs
- Thread creation done in the user space
- Bulk of thread scheduling and synchronization done in user space
- KLT's may be assigned to processors
- Combines the best of both approaches



"Many-to-Many" model

Advantages

- Flexible: OS creates kernel threads for physical concurrency

Applications creates user threads for application concurrency

Disadvantages

- Complex: Most use 1:1 mapping anyway

- Problem:
  - ✓ Thread creation: costly
  - ✓ And, the created thread exits after serving a request
  - ✓ More user request More threads, server overload
- Solution: thread pool
  - ✓ Pre-create a number of threads waiting for work
  - ✓ Wake up thread to serve user request --- faster than thread creation
  - ✓ When request done, don't exit --- go back to pool
  - ✓ Limits the max number of threads

Bibliography

- Silberschatz, A, Galvin, P.B, and Gagne, G., Operating System Principles, 9e, John Wiley & Sons, 2013.
- Stallings W., Operating Systems-Internals and Design Principles, 7e, Pearson Education, 2014.
- Harvey M. Deital, “Operating System”, Third Edition, Pearson Education, 2013.
- Andrew S. Tanenbaum, “Modern Operating Systems”, Second Edition, Pearson Education, 2004.
- Gary Nutt, “Operating Systems”, Third Edition, Pearson Education, 2004.

Acknowledgements

❖ I have drawn materials from various sources such as mentioned in bibliography or freely available on Internet to prepare this presentation.

❖ I sincerely acknowledge all sources, their contributions and extend my courtesy to use their contribution and knowledge for educational purpose.

Thank You!!

?