Threads, SMP, and Microkernels

Chapter 4

Current View of Process

- Process is a program in execution
- It has
  - Execution environment
    - address space, registers, etc
  - Execution entity
    - Code

- Currently thought of as a singular unit

Current View of a Process: Two Aspects

- *Resource ownership* - process includes a virtual address space to hold the process image
- *Scheduling/execution*- follows an execution path that may be interleaved with other processes

- However, these two characteristics are considered independently by the OS

Rethinking the “Process”

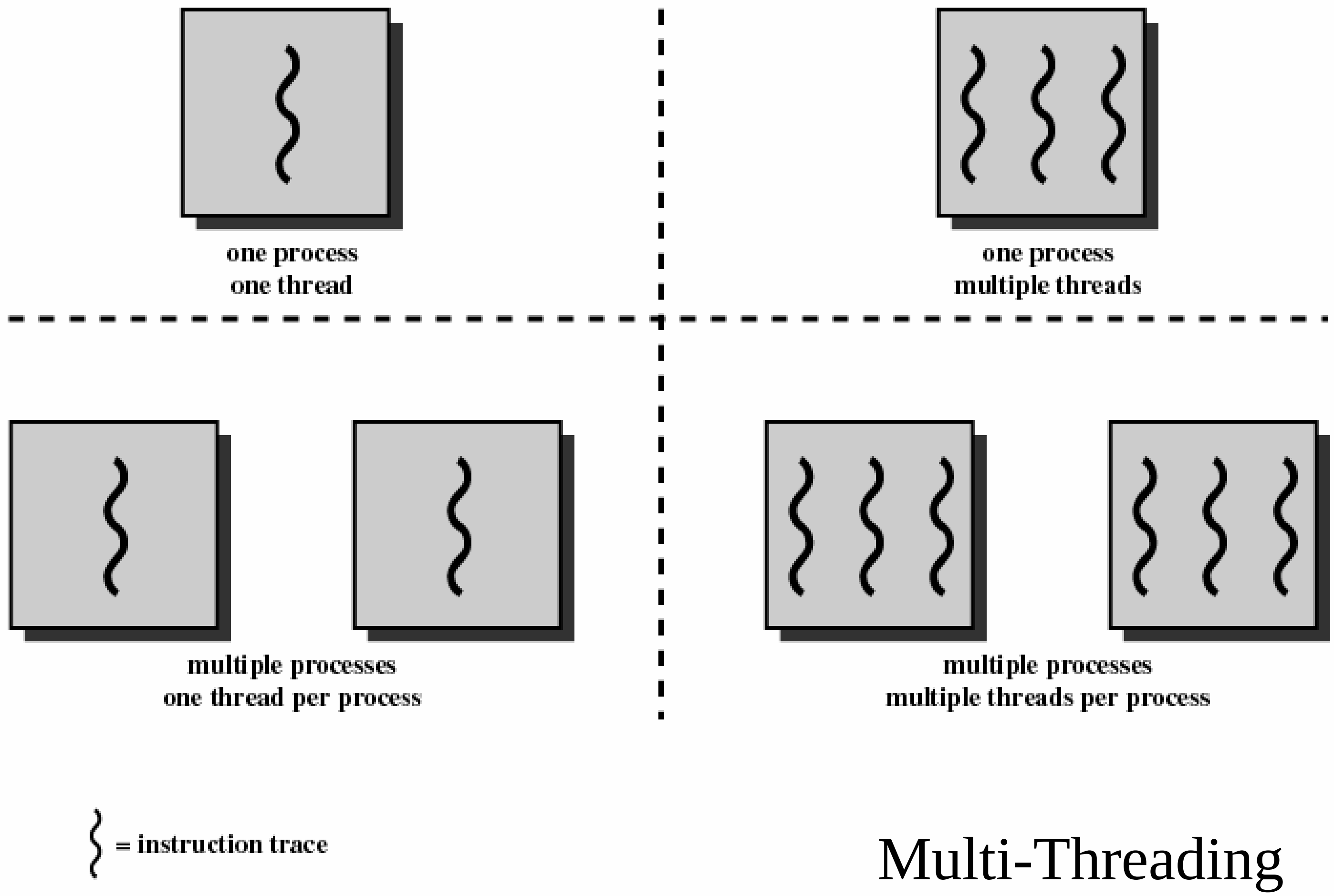
- Thread - Unit of dispatching
  - Computational entity +
  - Thread-specific memory

- Process – Execution environment
  - Threads
  - Resources available to all threads
    - Memory, files

Multithreading

Multiple threads of execution within a single process

- MS-DOS supports a single thread
- UNIX supports multiple user processes but only supports one thread per process
- Windows, Solaris, Linux, Mach, and OS/2 support multiple threads within a process



Multi-Threading

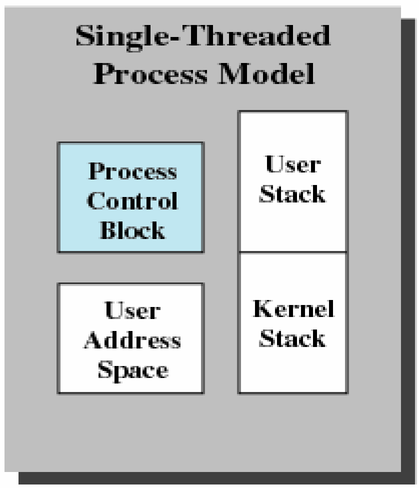
Process? / Thread?

- Is there a difference in the way we NOW think about them?

  => YES!

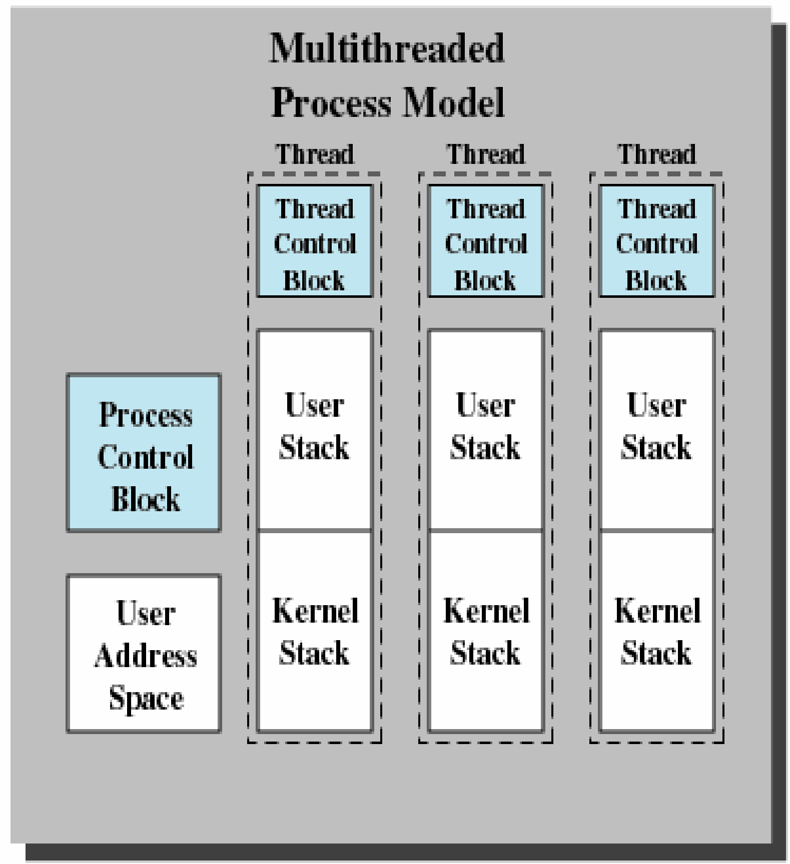
- Loosely speaking
  - Thread is the computational unit
  - Process is the resources allocated to the thread, i.e., it's computational environment,
    - Well… almost
  - Threads execute within, and are considered elements of a process

Process – Earlier Perspective



Process =
Computational unit +
Computational Environment

Process / Thread –
New Perspective



Thread

- Has an execution state (running, ready, etc.)
- Thread context saved when not running
- Has an execution stack
- Has some per-thread static storage for local variables

- Access to the memory and resources of its process
  – all threads of a process share this

Process

- Have a virtual address space which holds the process image
  - Process Control Block
  - User address space
    - Thread accessible
  - Thread + thread components *
- Has protected access to processors, other processes, files, and I/O resources
  - Viz-a-viz the OS

Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads *within the same process*
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

Uses of Threads in a Single-User Multiprocessing System

- Foreground to background work
- Asynchronous processing
  - Computation + polling
- Speed of execution
  - Computation + I/O
- Modular program structure
  - threads ⇔ functions

Process Implications w.r.t Threads

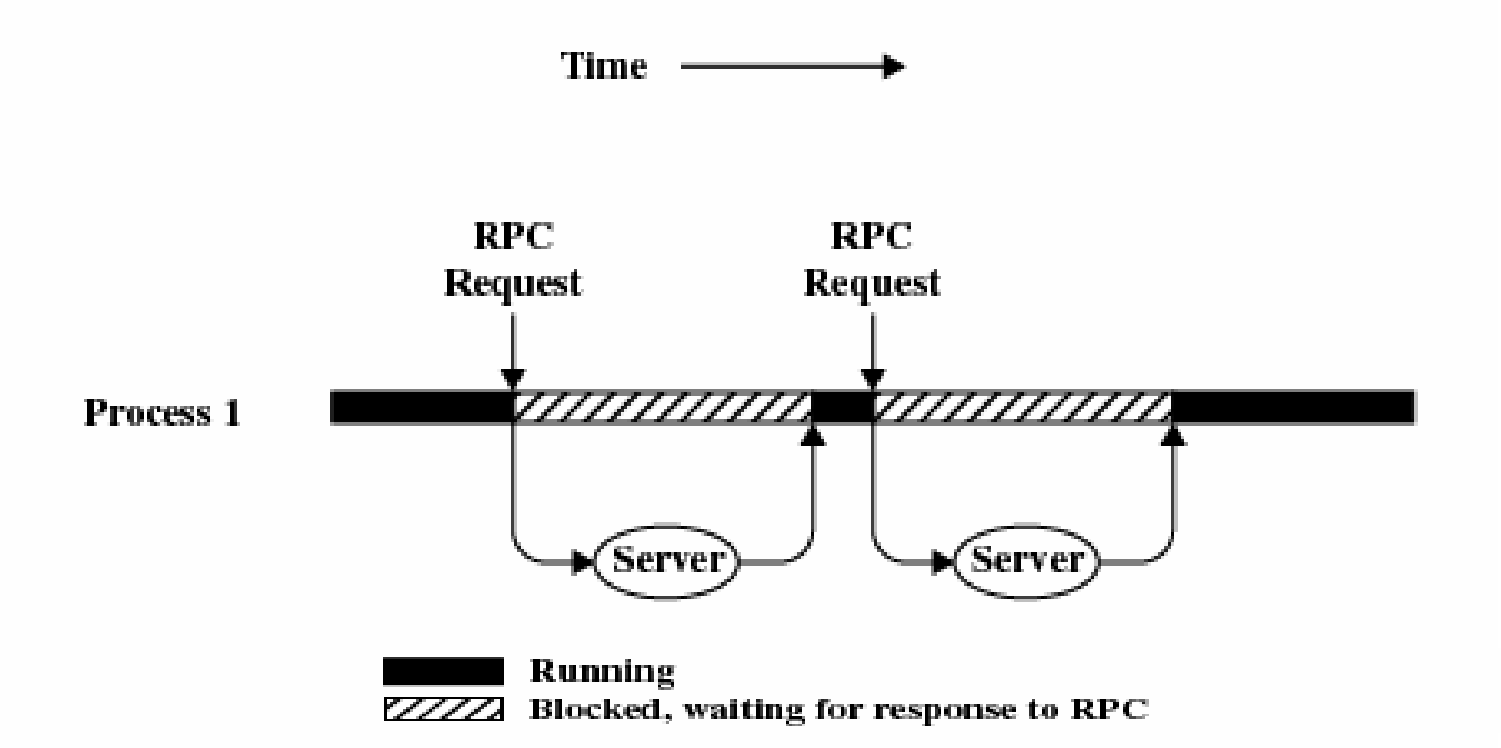
- Suspending a process involves suspending all threads of the process since all threads share the same address space
  - Does blocking a thread stop the process, and subsequently, all other processes?
    - ULT / KLT

- Termination of a process, terminates all threads within the process

Thread States

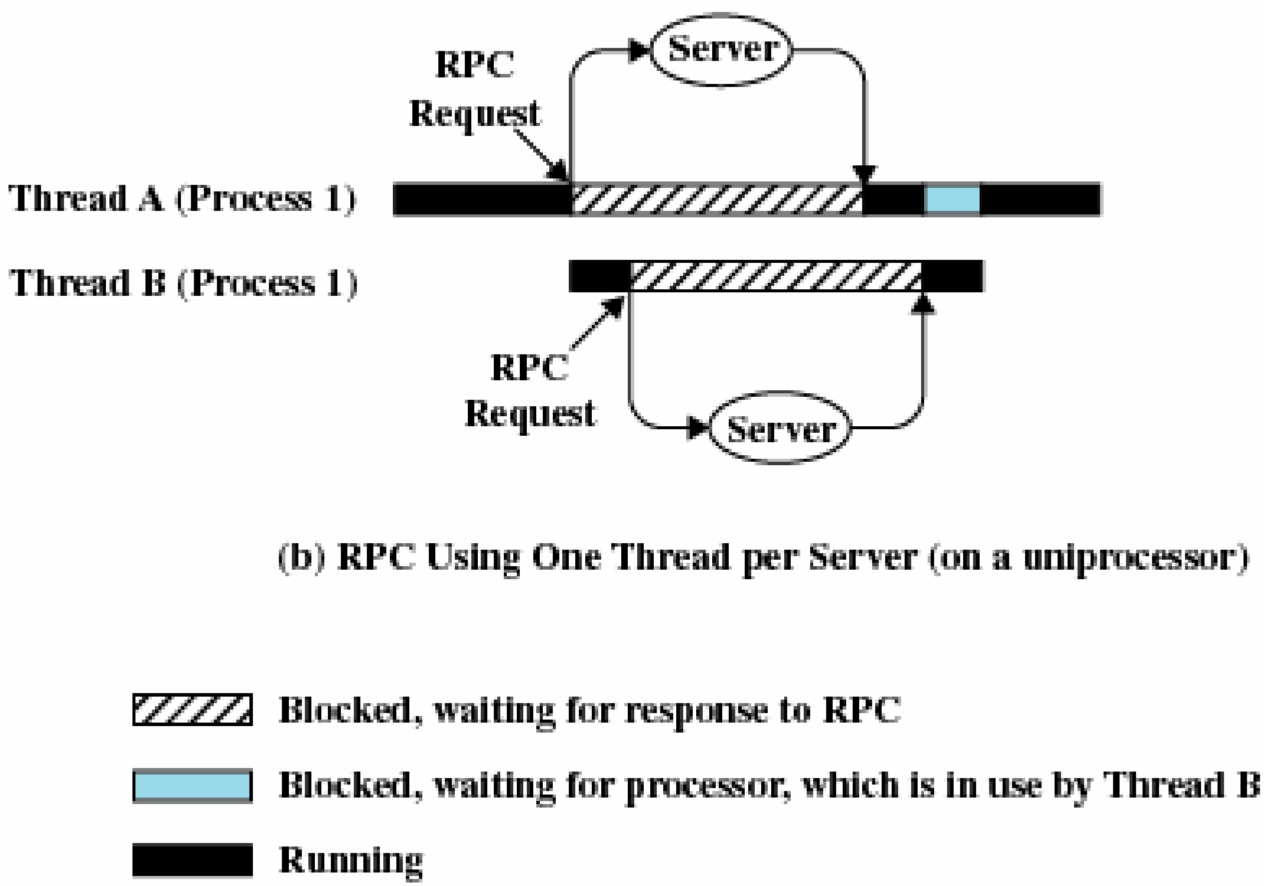
- States associated with a change in thread state
  - Spawn
    - Spawn another thread
  - Block
  - Unblock
  - Finish
    - Deallocate register context and stacks

Remote Procedure Calls Using a Single Threaded Process



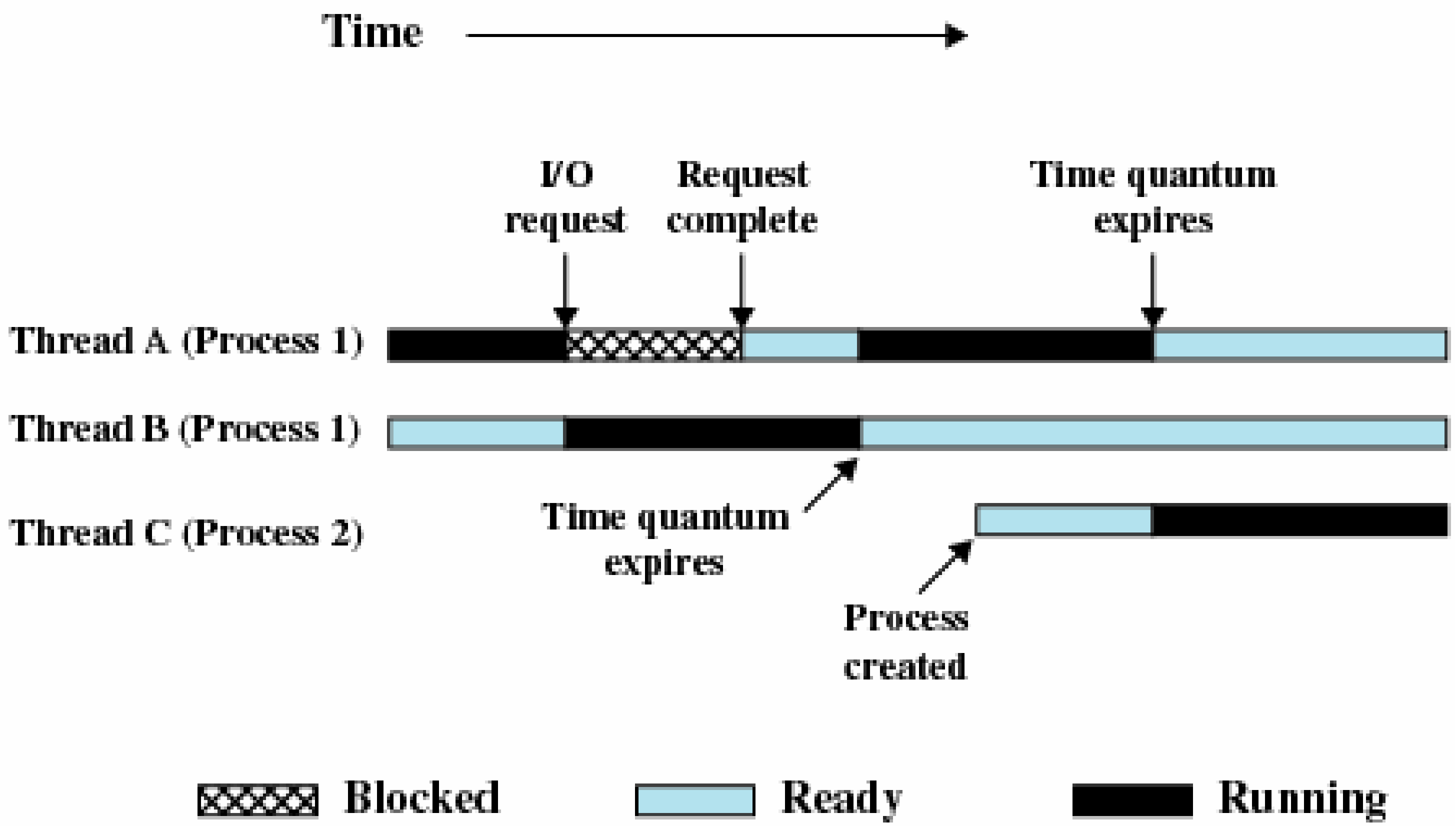
Remote Procedure Calls *Serialized*

Remote Procedure Call Using a Multi-Threaded Process

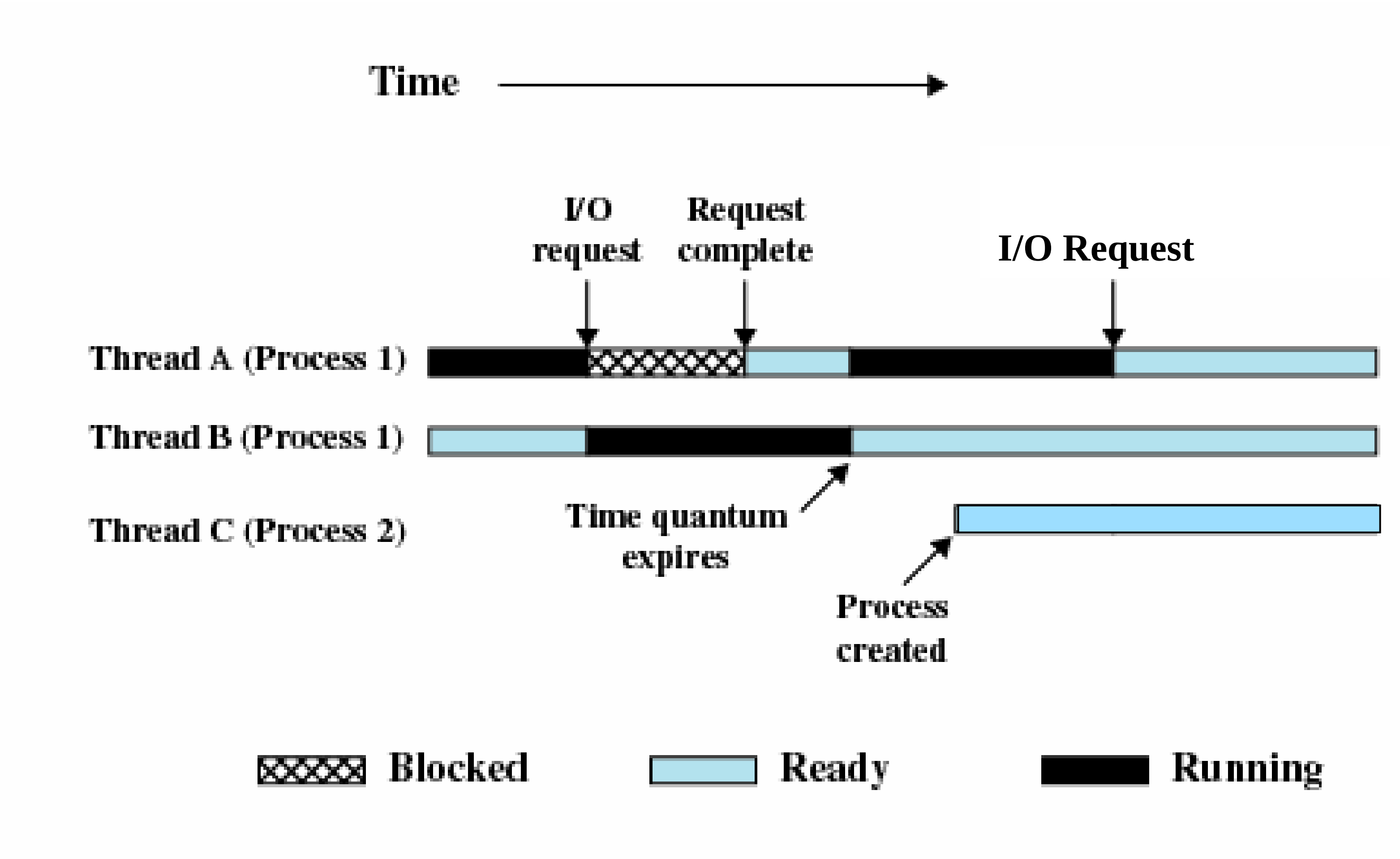


(b) RPC Using One Thread per Server (on a uniprocessor)

Multithreading / MultiProcessing



Multithreading / MultiProcessing



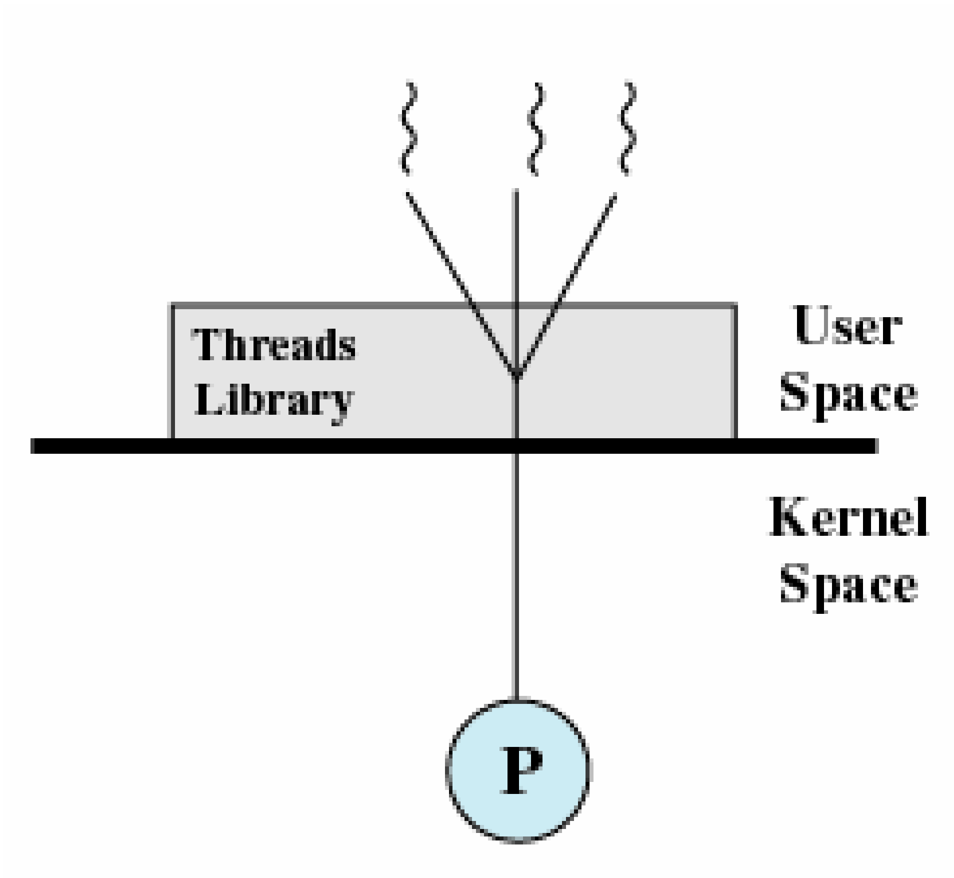
Who Should Get The Processor?

User-Level vs. Kernel-Level Threads

- User-Level
  - OS Not aware of their existence
- Kernel-Level
  - OS IS Aware of their existence

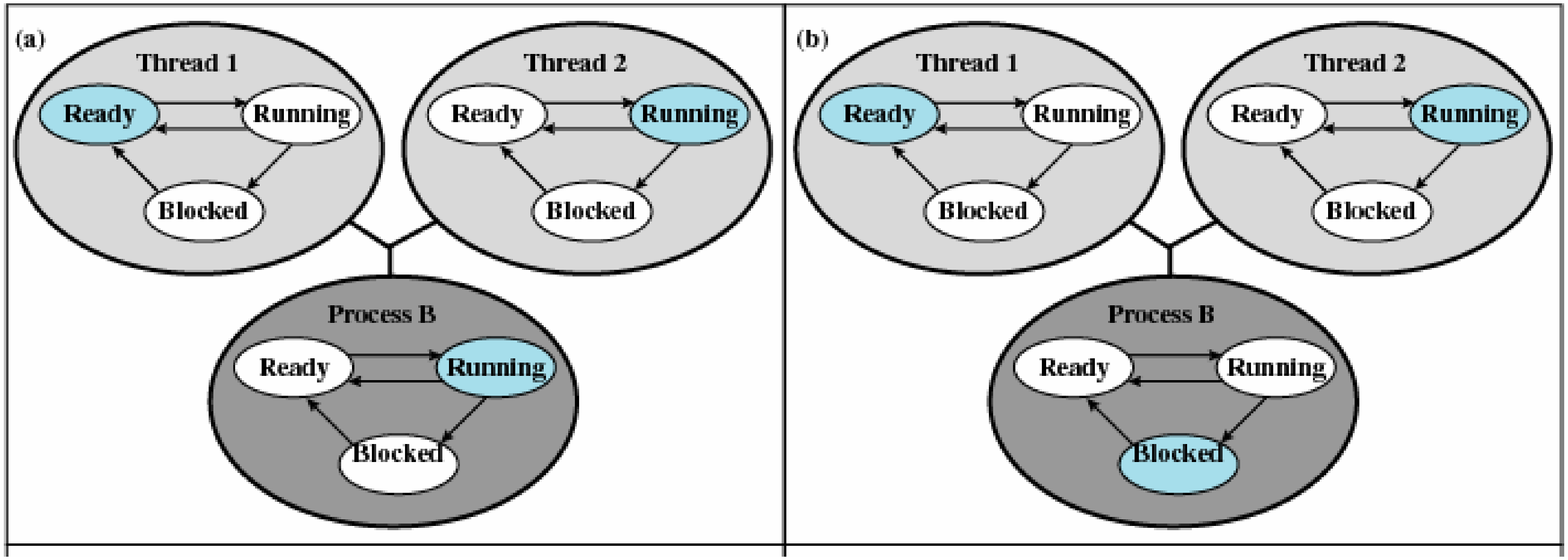
- Considerations
  - Who Schedules them for execution?
  - Time Quantum allocation
    - At Process or Thread level?
  - Does Thread block cause Process to block?

User-Level Threads



All thread management is done by the application

The kernel is not aware of the existence of threads

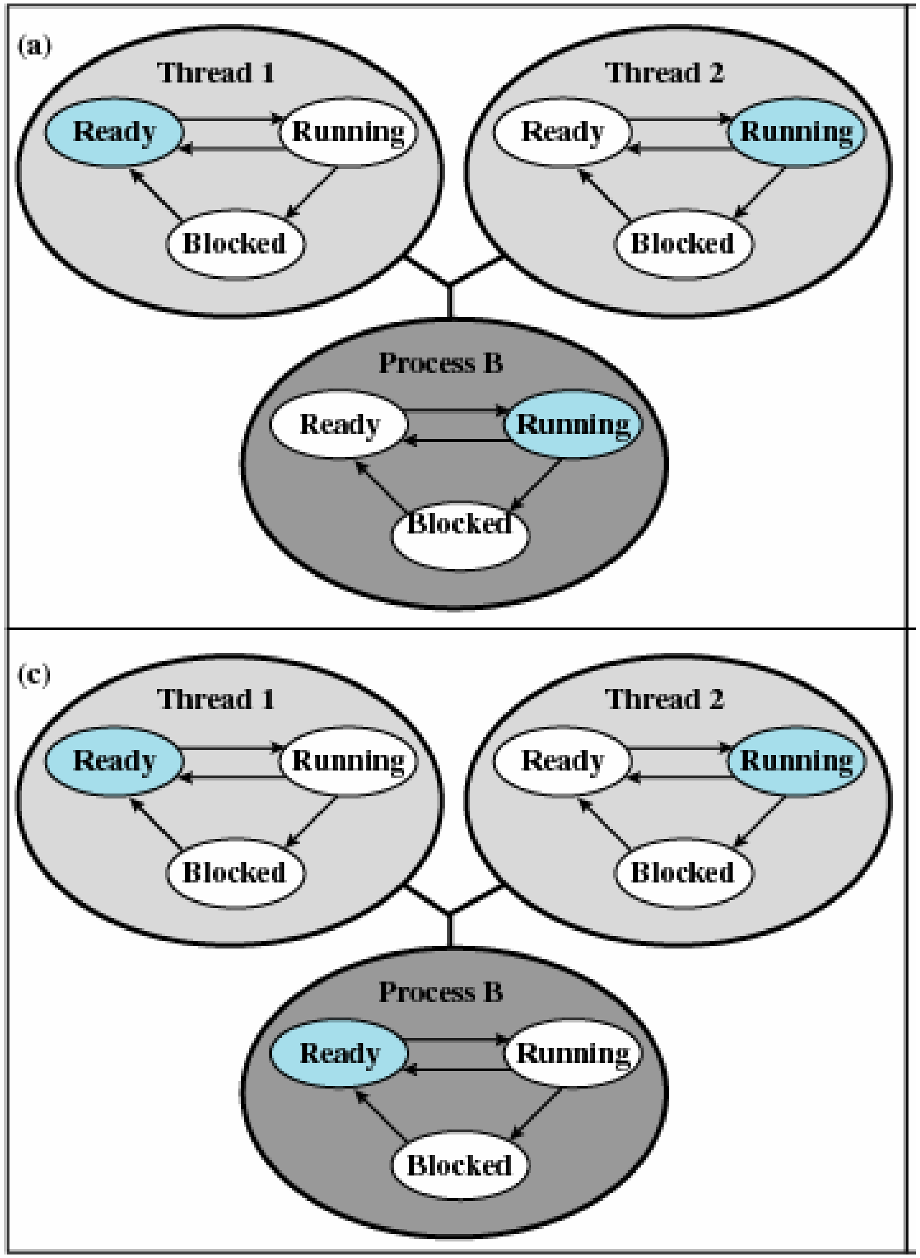


OS: Process B is executing
Application: Thread 2 is executing

Thread 2 requests I/O
OS perceives request from Process
OS Blocks **Process B**

Note: Thread 2 still in "running" State!

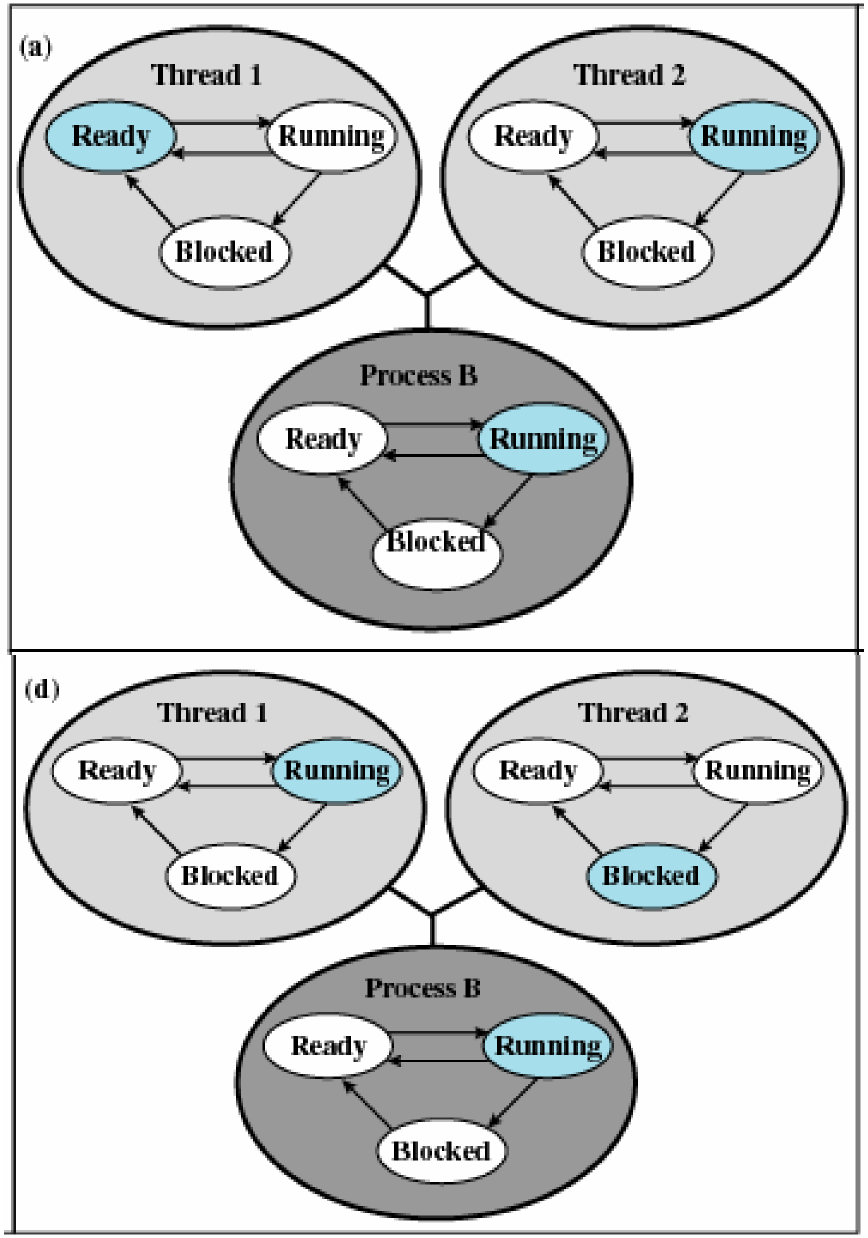
ULTs *explicitly* issue block or yield to change states



OS: Process B executing
App: Thread 2 executing

Quantum up for Process B
OS: Process B => Ready

Note:
  Thread 2 still in running state



OS: Thread B executing
App: Thread 2 executing

Thread 2 intentionally issues block

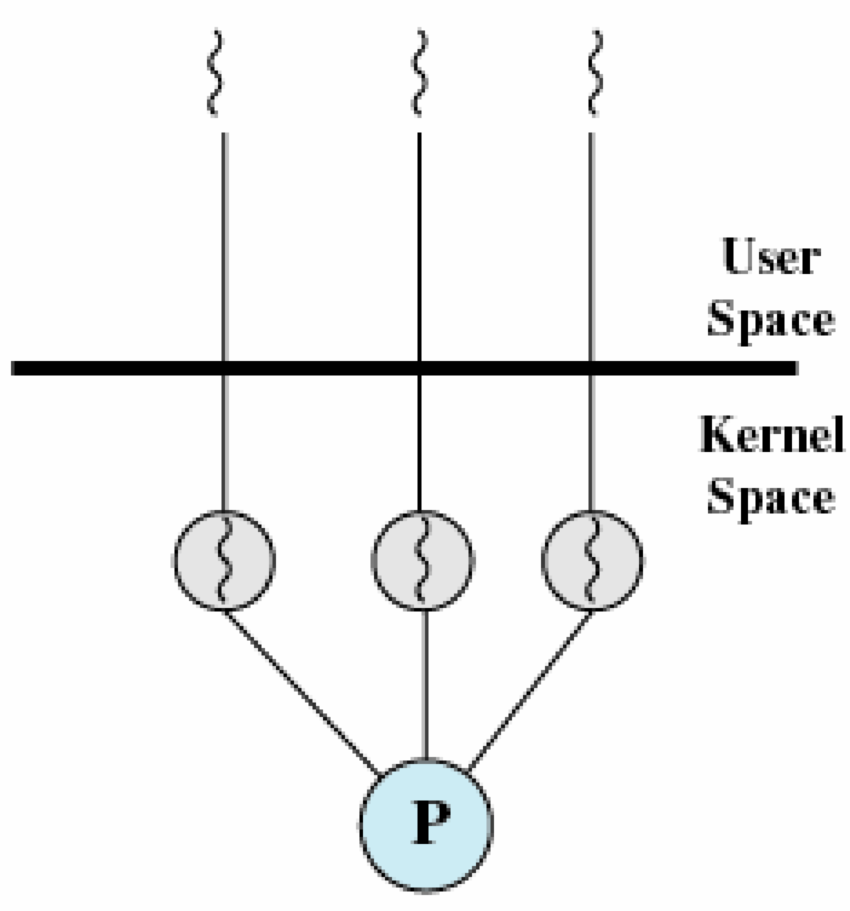
ULT Lib:
- Thread 2 => Blocked State
- Thread 1 => Running State

OS: Thread B still running
App: Thread 1 executing

ULKs: The Good , The Bad

- Advantages
  - Thread level switching does not require kernel mode privildges (no Mode switching)
  - Scheduling can be application specific
  - ULT’s can run on any OS
- Disadvantages
  - If a thread issues a system-level call that blocks thread, then entire Process blocks
  - Cannot take advantage of Multiprocessor environment, e.g. SMP

Kernel-Level Threads



Kernel maintains context information for *both* the process and the threads

Kernel (OS) schedules each thread individually

Windows uses this approach

KLT: The Good, The Bad

- Advantages
  - Thread management done by OS Kernel
  - Scheduling at thread level, not process level
  - In a multiprocessor environement we can have true concurrency
  - If a thread issues a blocking system call, the other threads are not affected
- Disadvantages
  - Transfer of control form one thread to another expensive
    - Two Mode switches (U->K, K->U) : Context switch

User-Level vs. Kernel-Level Threads (Revisited)

- User-Level: OS Not aware of their existence
- Kernel-Level: OS IS Aware of their existence

- Considerations
  - Who Schedules them for execution?
  - Time Quantum allocation
    - At Process or Thread level?
  - Does Thread block cause Process to block?

Operational Overhead: ULK vs KLT

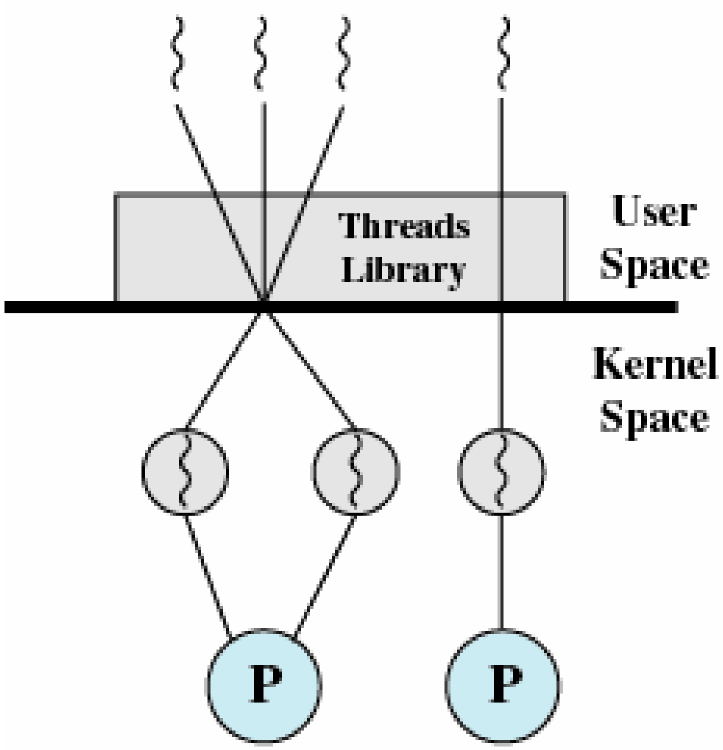
Table 4.1 Thread and Process Operation Latencies (μs) [ANDE92]

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Null Fork: OH of creating a thread
Signal Wait: OH in synchronizing two process/thread together

Implications: KLTs are expensive

Combined Approaches Do Exist



(c) Combined

SUN Solaris

Process created with single ULT thread running in user space

Additional ULT threads created in user space

ULTs are then mapped (transformed) into KLT – controlled by application programmer

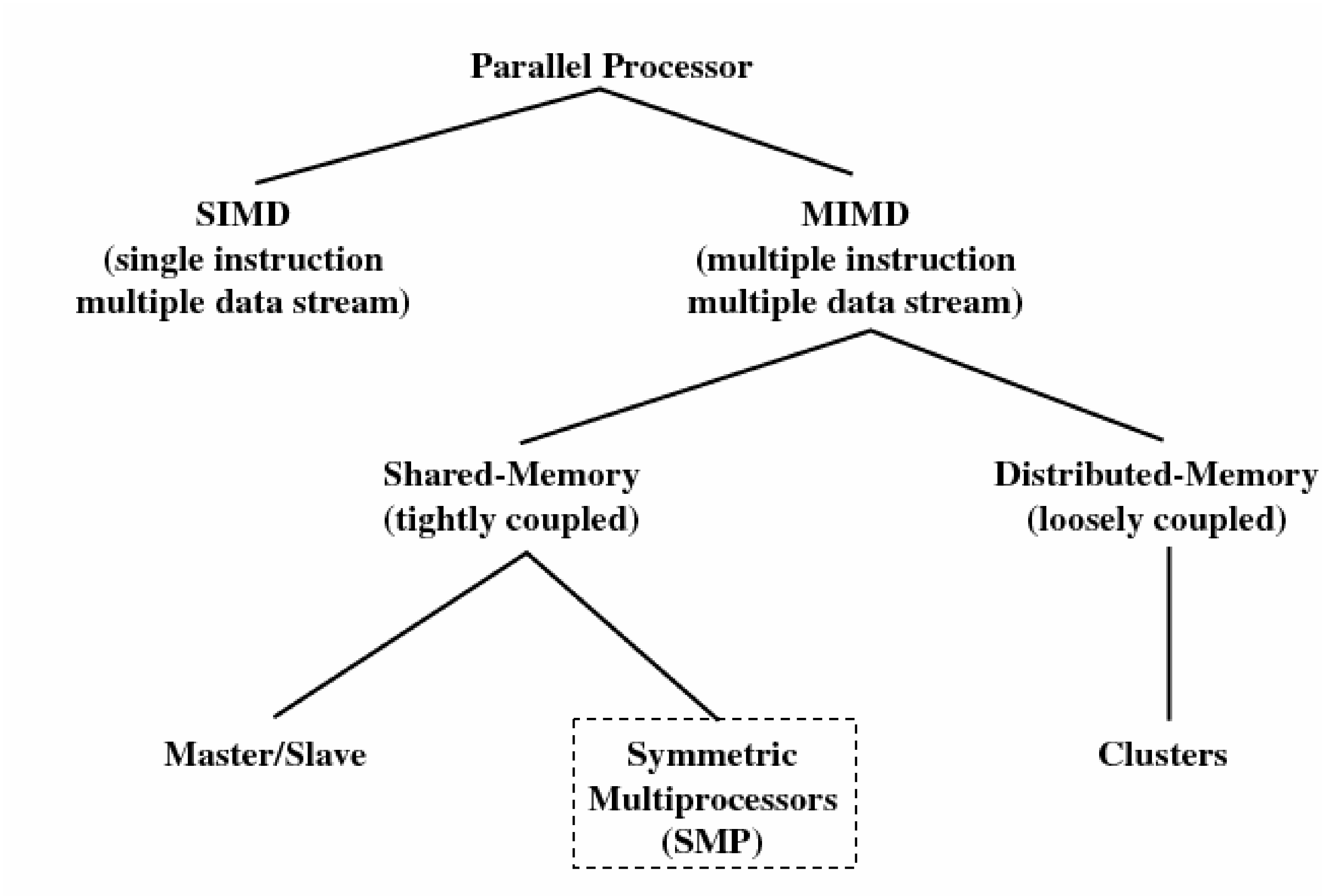
Categories of Computer Systems

- Single Instruction Single Data (SISD) stream
  - Single processor executes a single instruction stream to operate on data stored in a single memory
- Single Instruction Multiple Data (SIMD) stream
  - Each instruction is executed on a different set of data by the different processors

Categories of Computer Systems

- Multiple Instruction Single Data (MISD) stream
  - A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. Never implemented
- Multiple Instruction Multiple Data (MIMD)
  - A set of processors simultaneously execute different instruction sequences on different data sets

Parallel Processors: SIMD / MIMD



Symmetric Multiprocessing

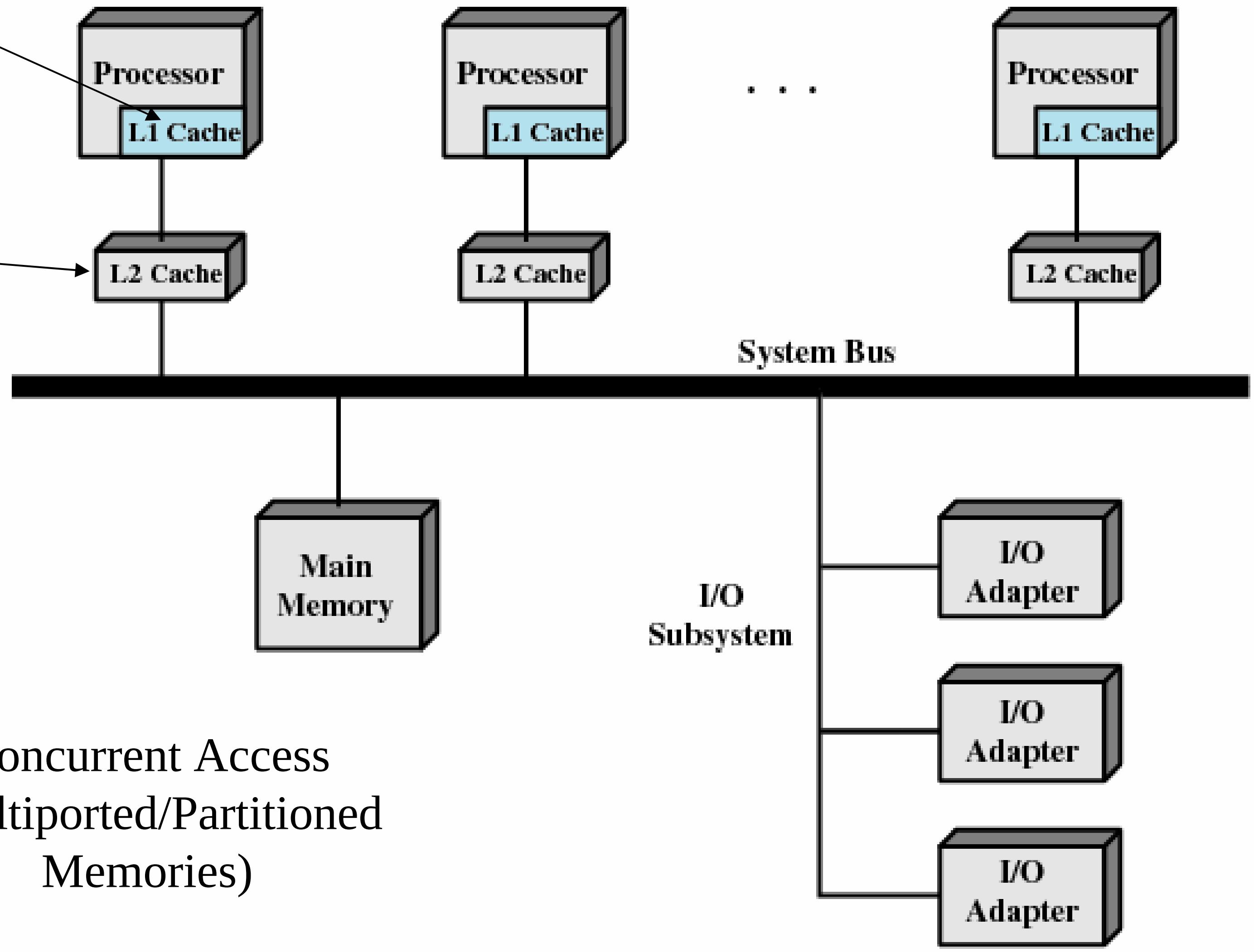
- Kernel can execute on any processor
- Kernel can be constructed as multiple processes/threads and execute concurrently

- Typically each processor does self-scheduling from the pool of available process or threads

Memory & Cache Organization

On Processor Chip
(fastest)

On Motherboard
(Faster than accessing
memory)



Concurrent Access
(Multiported/Partitioned
Memories)

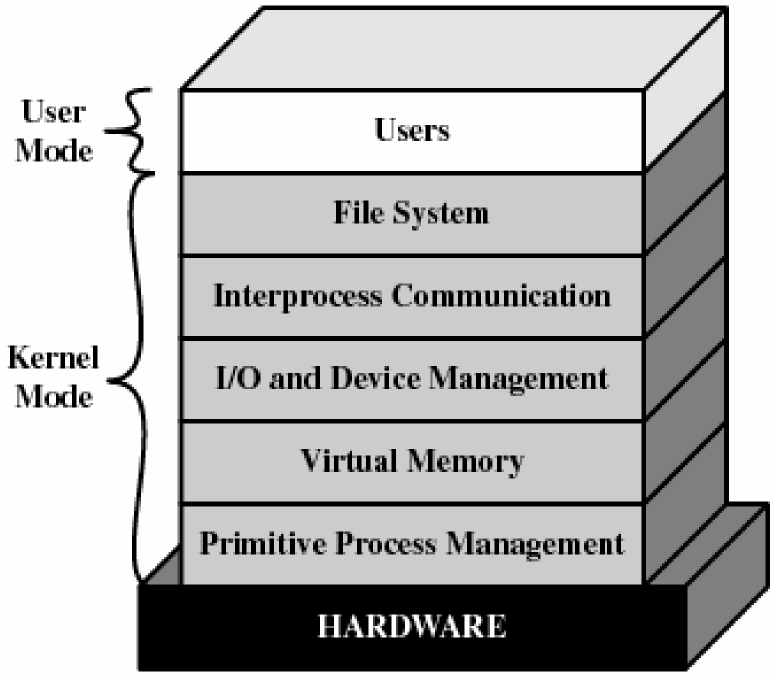
Multiprocessor Operating System Design Considerations

- Kernel processes need to be re-entrant
  - Simultaneous concurrent processes or threads
- Scheduling can be performed by more than one processor
  - Need to avoid conflicts
- Synchronization
  - Facility for mutual exclusion & event sequencing
- Memory management
  - Concurrent access
- Reliability and fault tolerance
  - Graceful degradation if one processor fails

OS “Kernels”

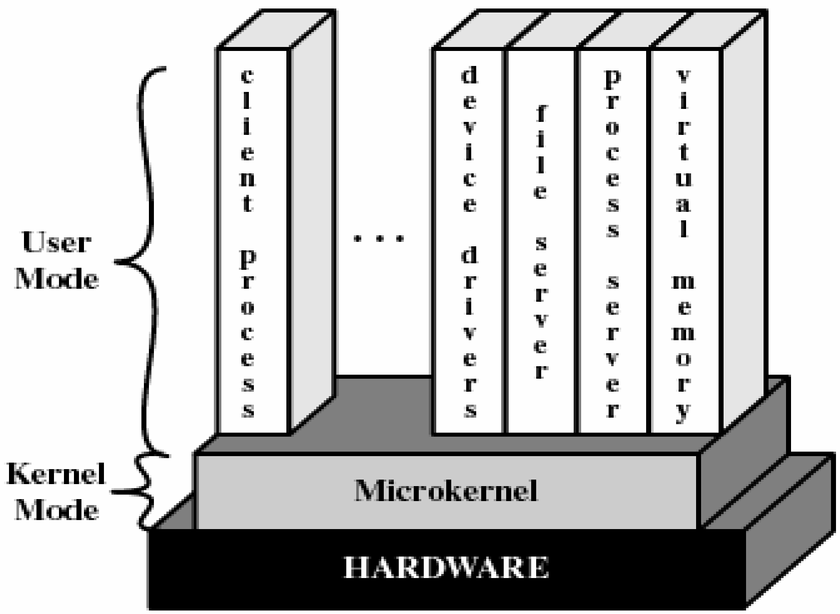
- Monolithic
  - Lacked structure
  - Any procedure could call any other
  - OS/360 1Mill SLOC, Multics 20 Mill Slocs
- Layered
  - Structured, but everything still ran in Kernel mode
- Microkernels
  - Only essential run in Kernel mode
  - Remainder ran as services

Layered Kernel



- Hierarchically organized
- Interaction between adjacent layers
- Most layers executed in Kernel mode
- Modifying code still a problem
- Security difficult (so many interfaces)

Microkernels



(b) Microkernel

- Small operating system core
- Contains only essential core OS functions
- Many traditional OS services now external subsystems
  - Device drivers
  - File systems
- Services implemented as server processes
  - Message passing

Benefits of a Microkernel Organization

- Uniform interface on request made by a process
  - Don't distinguish between kernel-level and user-level services
  - All services are provided by means of message passing
- Extensibility
  - Allows the addition of new services
- Flexibility
  - New features easily added
  - Existing features can be subtracted

Benefits of a Microkernel Organization

- Portability
  - Changes needed to port the system to a new processor is changed in the microkernel - not in the other services
- Reliability
  - Modular design
  - Small microkernel can be rigorously tested

Benefits of Microkernel Organization

- Distributed system support
  - Message are sent without knowing what the target machine is
- Object-oriented operating system
  - Components are objects with clearly defined interfaces that can be interconnected to form software

Microkernel Design

- Low-level memory management
  - Mapping each virtual page to a physical page frame

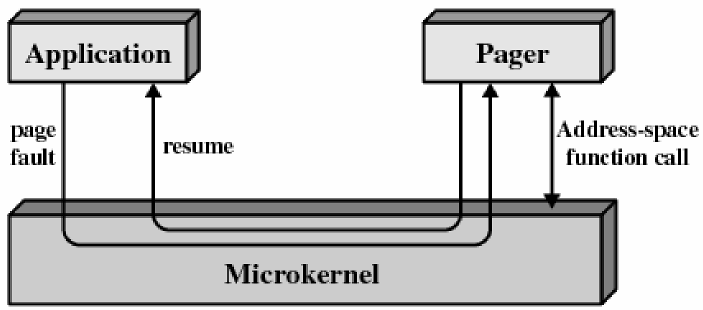


Figure 4.11 Page Fault Processing

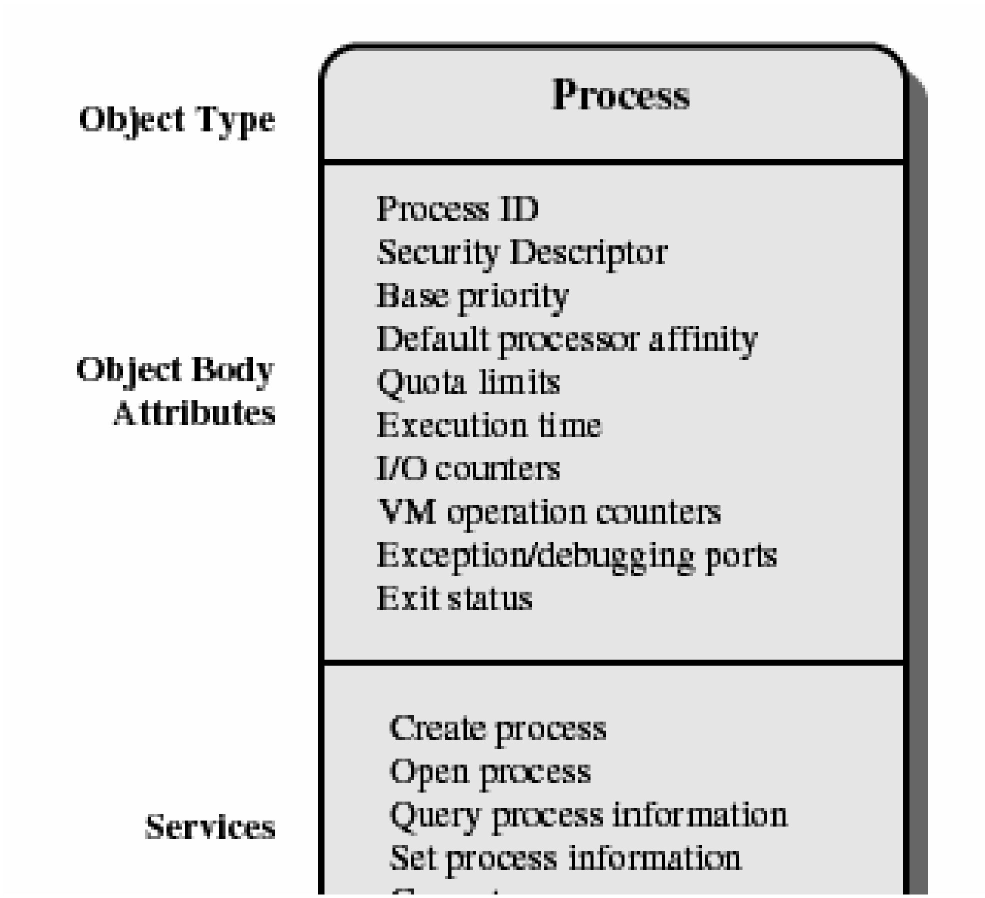
Microkernel Components

- Low-level memory management
  – Page fault initiates MK interupt
- Interprocess communication
  – Port-based communication
  – (sender, message)
- I/O and interrupt management

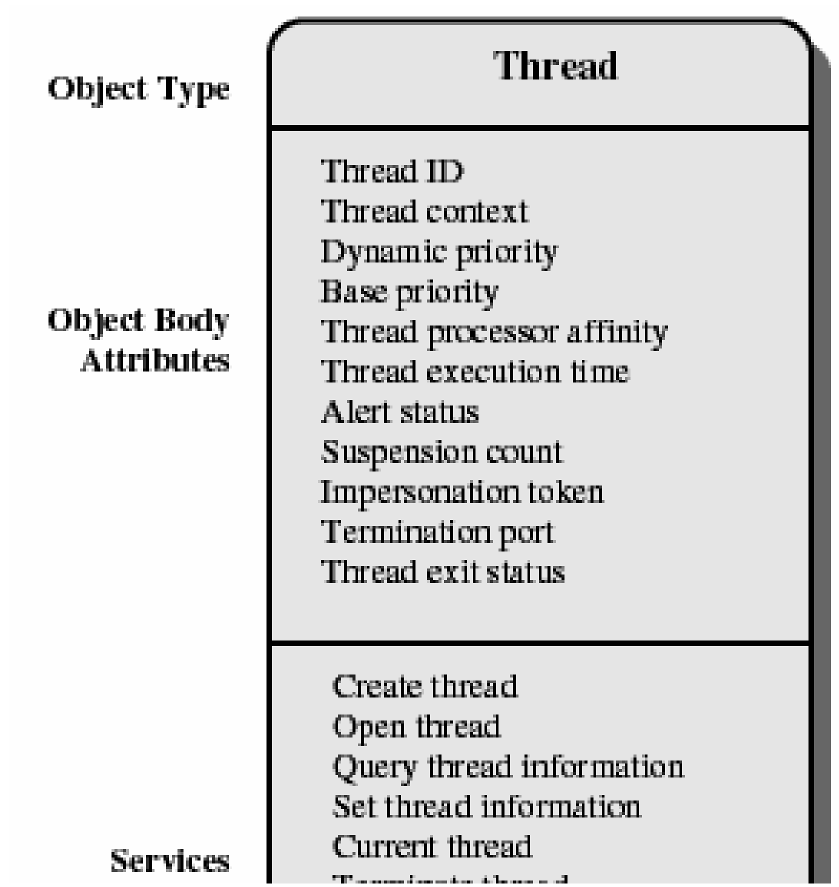
Windows Processes

- Process & Thread separate concepts
- Threads are kernel-based
- ULTs achieved through library calls
- An executable process may contain one or more threads
- Both processes and thread objects have built-in synchronization capabilities

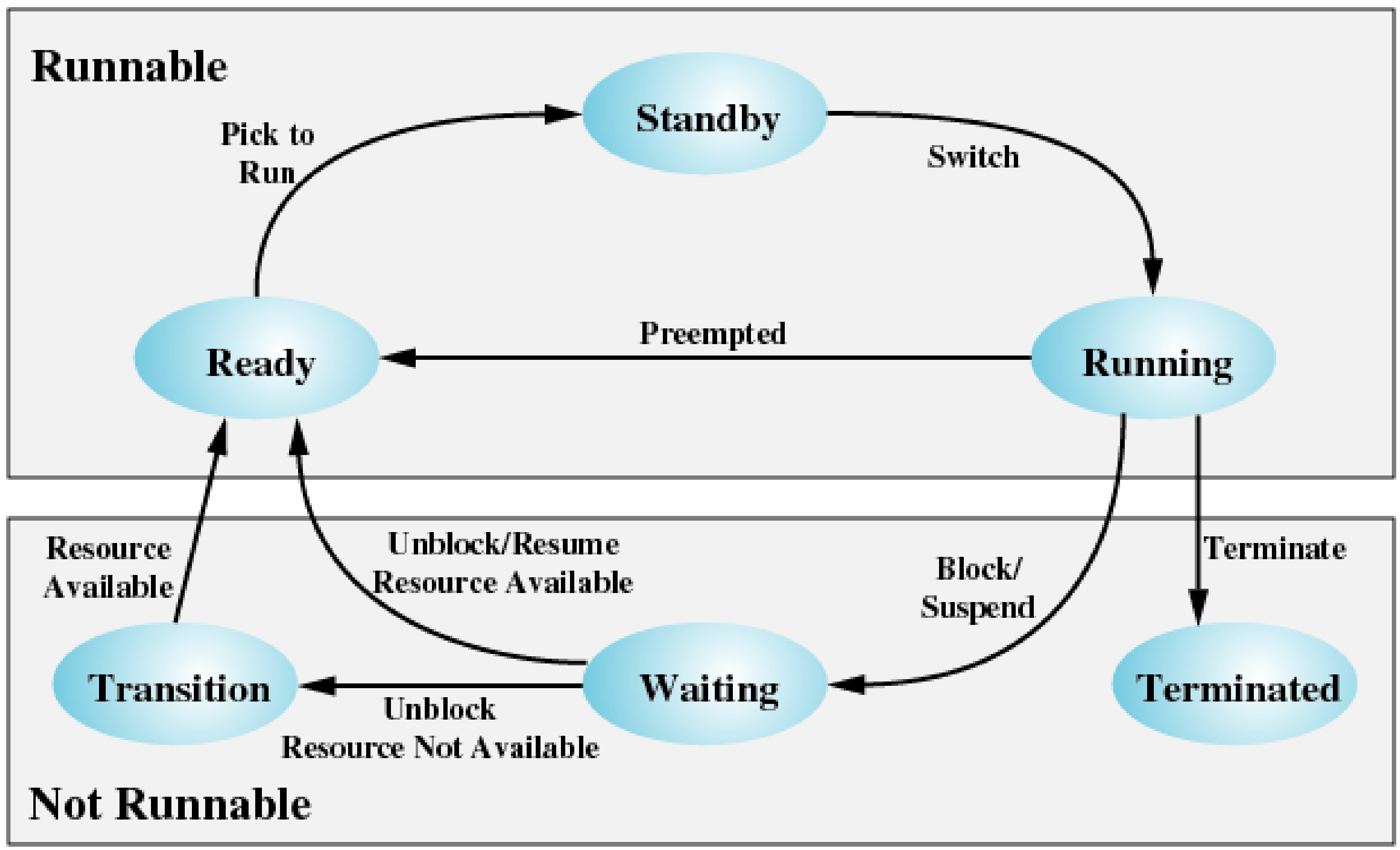
Windows Process Object



Windows Thread Object



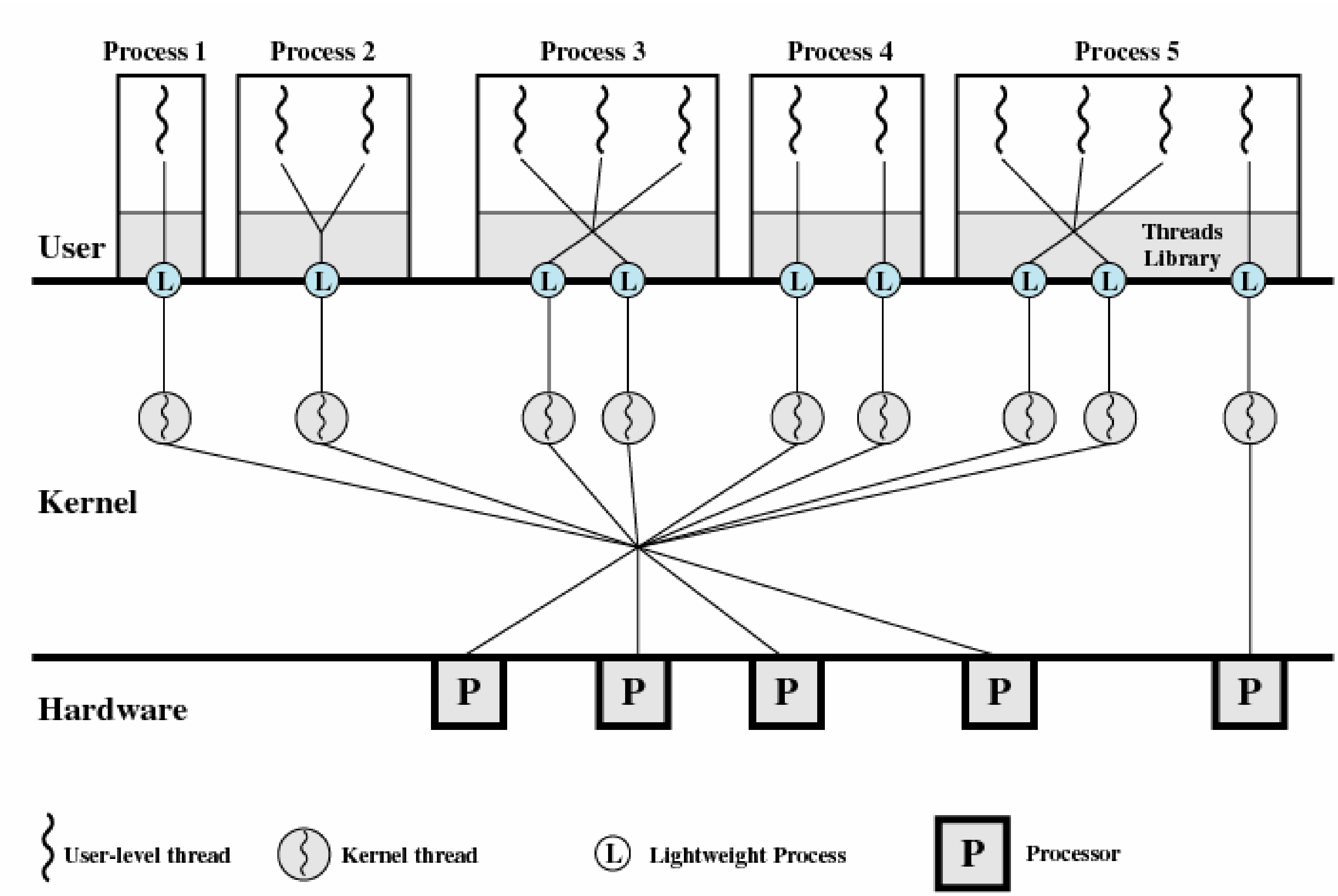
Windows Thread States



Solaris (SUN)

- Process includes the user's address space, stack, and process control block
- User-level threads
  - Library supported
- Lightweight processes (LWP)
  - Associates ULT with KLT
- Kernel threads

Traditional Unix	Pure ULT	Multiplexed ULTs	Pure "KLT"s	Combo



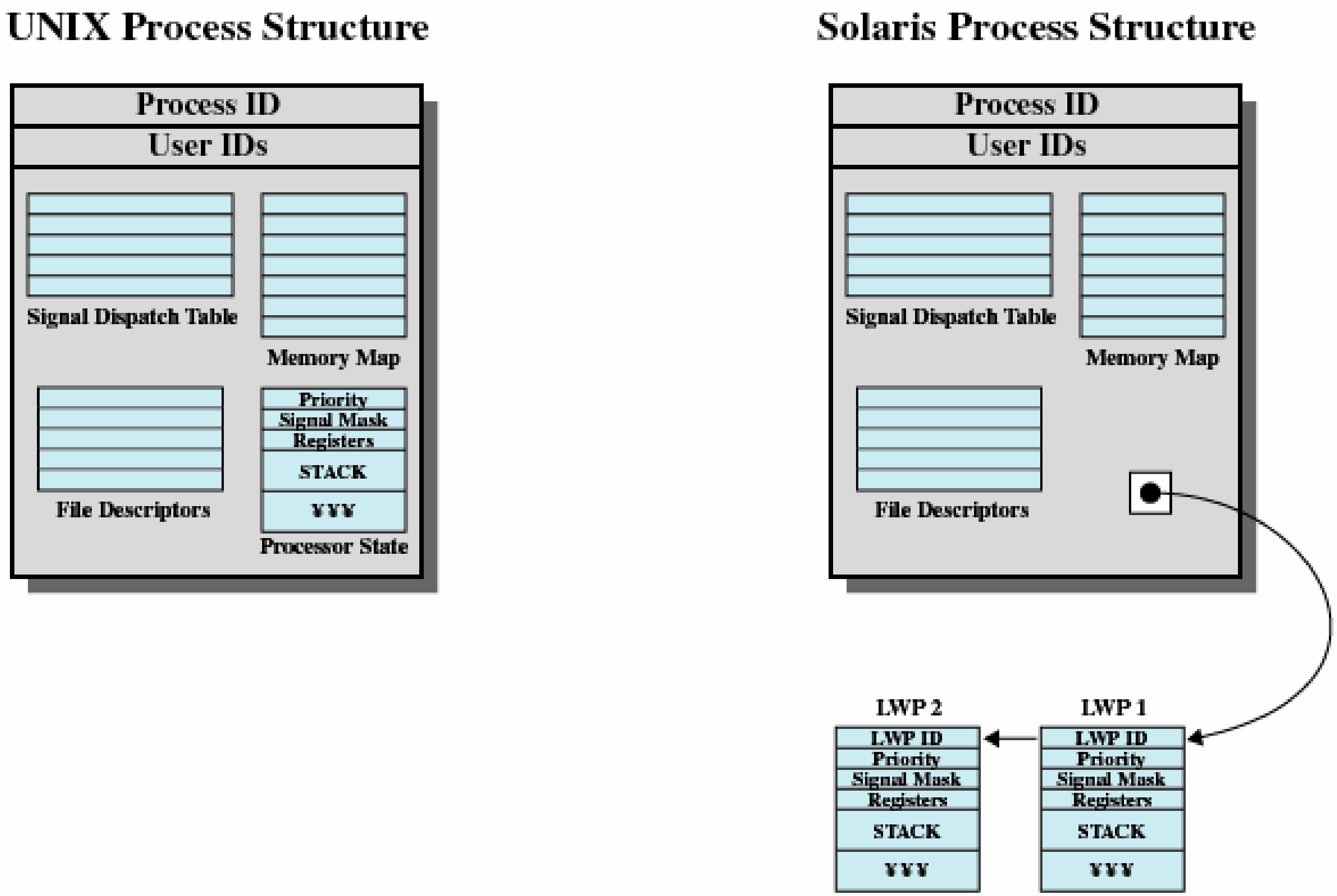


Figure 4.16 Process Structure in Traditional UNIX and Solaris [LEWI96]

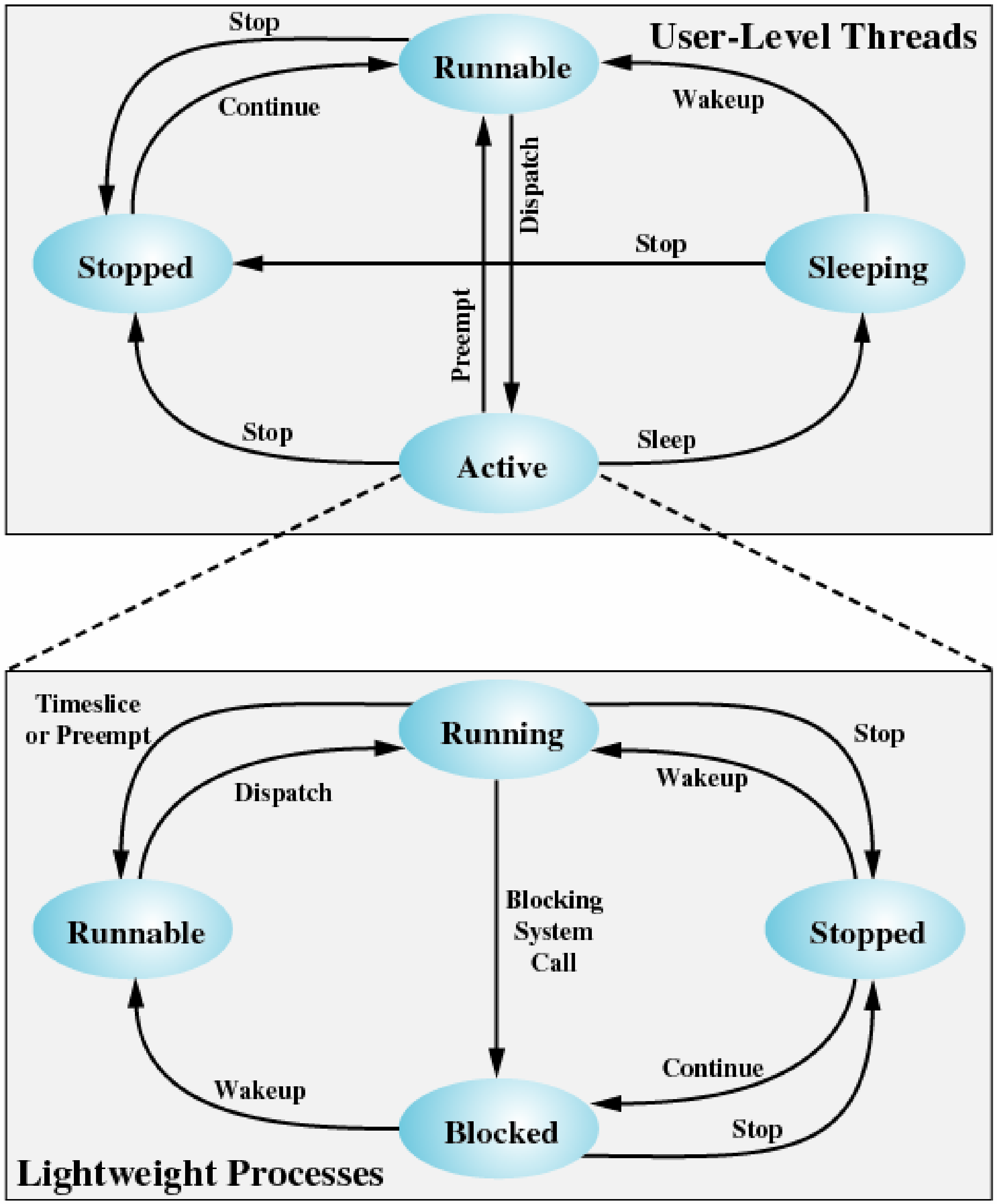


Figure 4.17 Solaris User-Level Thread and LWP States

Managed through application by calls to library routines

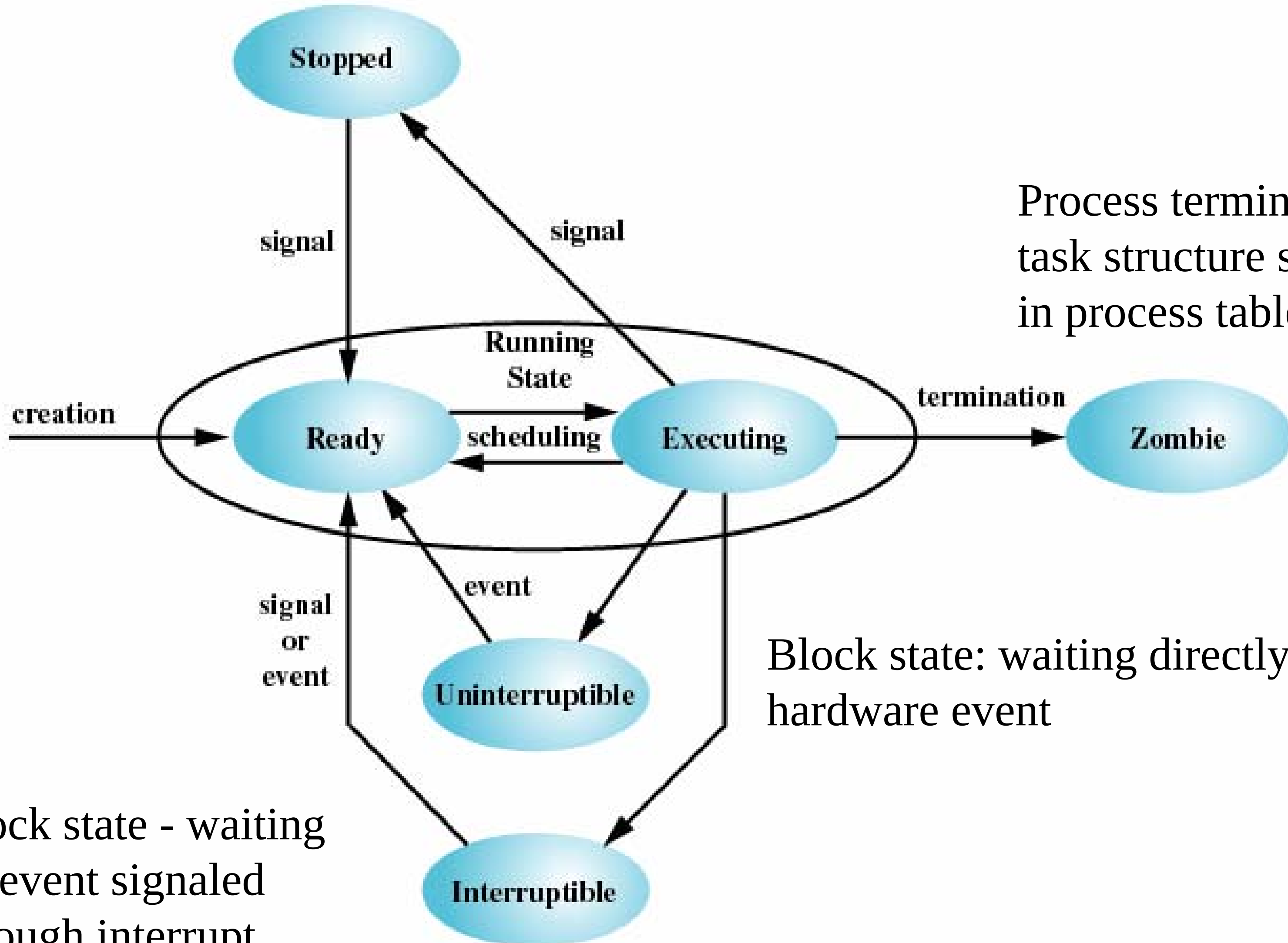
ULT can be in active state even if LWP is blocked – no computation occurs

Managed by OS Kernel

Linux Process/Thread

- Classical view
  - Process and Thread viewed as one entity
  - Fork()
    - creates “copy” of parent process
    - Separate address space
- Modern view
  - Multithreading
  - Clone()
    - Shares address space, resources, code
    - Individual thread stack, PSW

Linux Process/Thread Model



Process terminated, task structure still in process table

Block state: waiting directly on hardware event

Block state - waiting on event signaled through interrupt