Threads

Giorgio Richelli
giorgio_richelli@it.ibm.com

Contents

- Introduction
- Fundamental Abstractions
- Lightweight Process Design
- User-Level Threads Libraries

Giorgio Richelli
giorgio_richelli@it.ibm.com

Introduction

- Motivation
  - Sharing common resources
    - Possible with processes, but overheads do exist
    - *fork()* is expensive
  - Multiprocessor architecture
    - The *process* is fundamentally a “single CPU” abstraction

Giorgio Richelli
giorgio_richelli@it.ibm.com

Multiple Threads and Processors

- True parallelism for multiprocessor architectures
- Multiplex if #T > #P
- Ideally:
  - if an application need 1 unit time with one thread version, it will only need 1/n unit time with a multithread version on a computer with n processors.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Traditional UNIX system

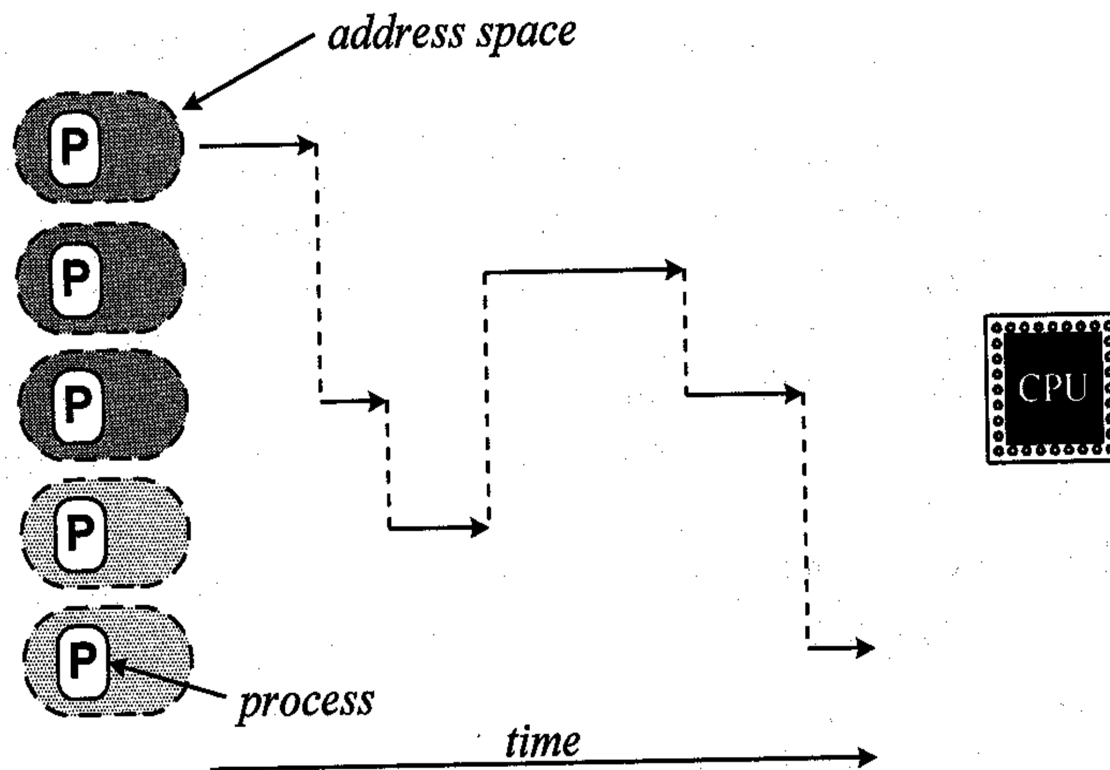


Figure 3-1. Traditional UNIX system—uniprocessor with single-threaded processes.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Multithreaded Processes

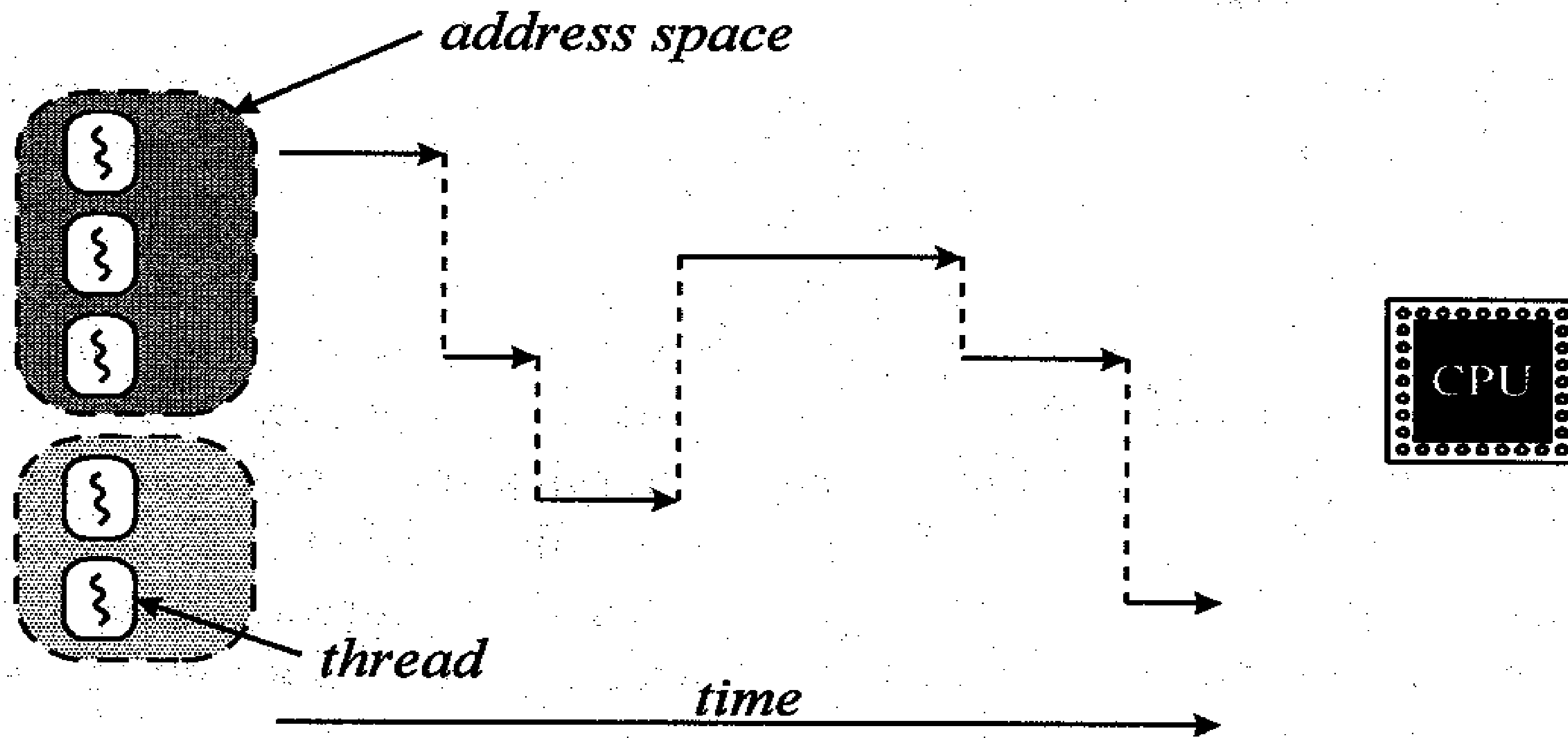


Figure 3-2. Multithreaded processes in a uniprocessor system.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Multiprocessor

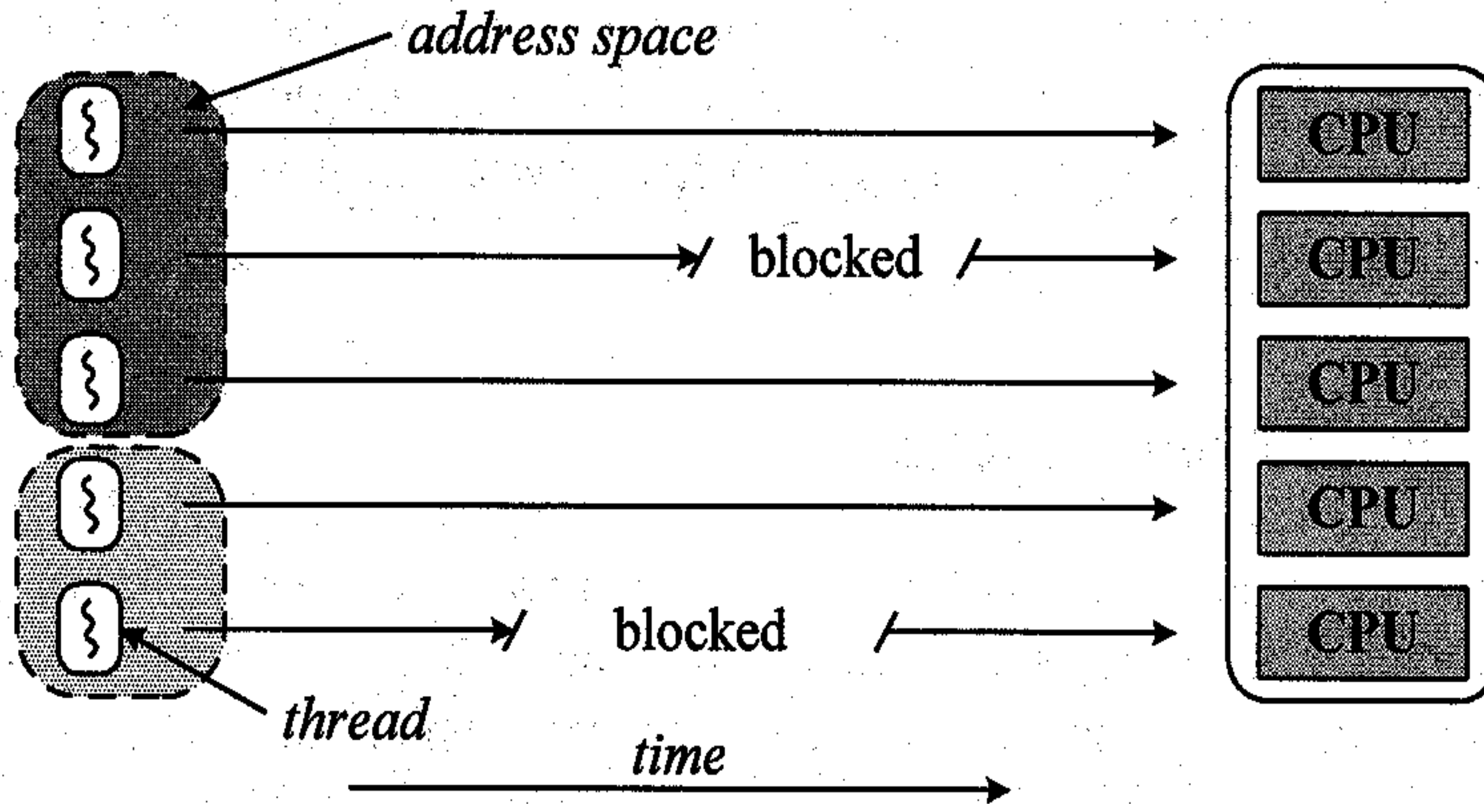


Figure 3-3. Multithreaded processes on a multiprocessor.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Concurrency & Parallelism

- Concurrency:
  - The *maximum parallelism* it can achieve with an *unlimited number* of processors.
- Parallelism:
  - The actual degree of parallel execution achieved and is limited by the number of physical processors.
- User/System concurrency
  - hot threads (kernel) /cold threads (coroutines)

Giorgio Richelli
giorgio_richelli@it.ibm.com

Fundamental Abstraction

- A *process* :
  - is a compound entity that can be divided into two components:
    - a set of threads
    - a collection of resources.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Fundamental Abstraction

- A *thread* :
  - a dynamic object that represents a control point in the process and that executes a sequence of instructions.
  - Shared resources
  - Private objects:

    pc, stack, register context

Giorgio Richelli
giorgio_richelli@it.ibm.com

Processes

- Have a virtual address space which holds the process image
- Protected access to processors, other processes, files, and I/O resources

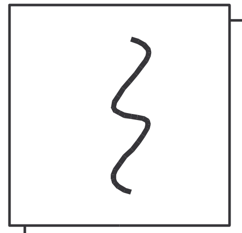
Giorgio Richelli
giorgio_richelli@it.ibm.com

Threads

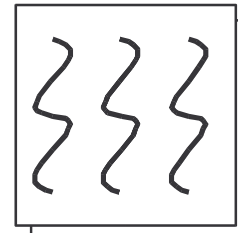
- Has an execution state (running, ready, etc.)
- Saves thread context when not running
- Has an execution stack
- Has some per-thread static storage for local variables
- Has access to the memory and resources of its process
  - all threads of a process share this

Giorgio Richelli
giorgio_richelli@it.ibm.com

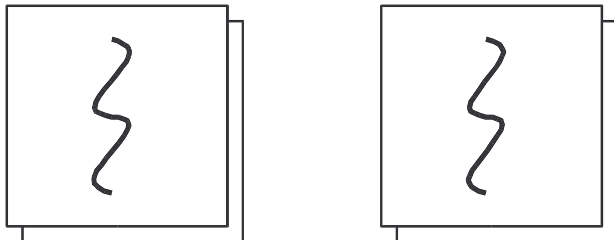
Threads and Processes



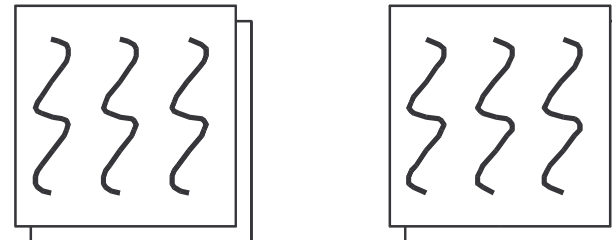
one process
one thread



one process
multiple threads



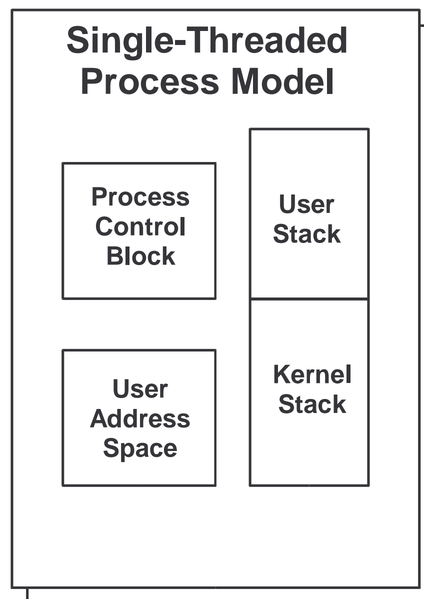
multiple processes
one thread per process

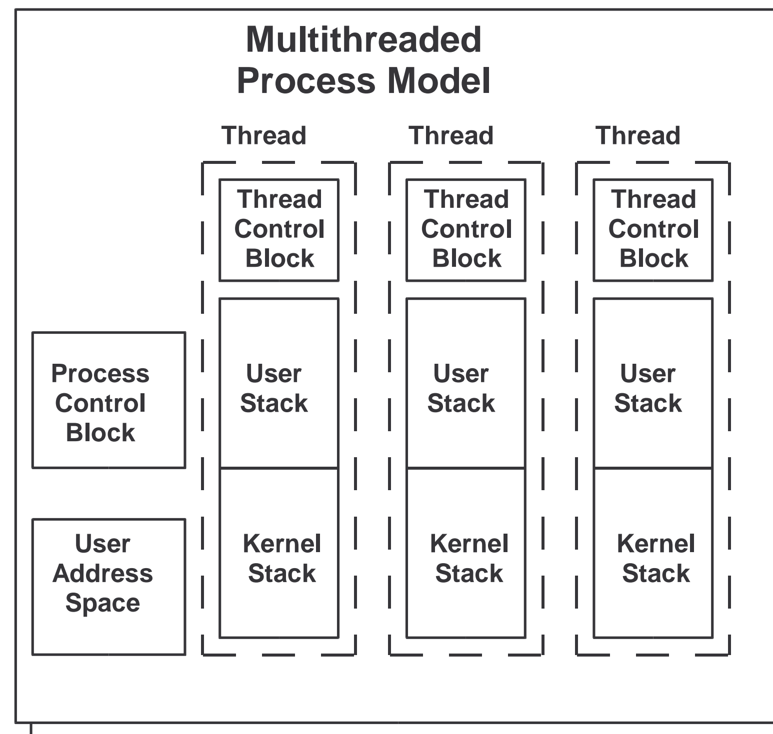


multiple processes
multiple threads per process

Giorgio Richelli
giorgio_richelli@it.ibm.com

Single Threaded and Multithreaded Process Models





Giorgio Richelli
giorgio_richelli@it.ibm.com

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

Giorgio Richelli
giorgio_richelli@it.ibm.com

Threads

- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process

Giorgio Richelli
giorgio_richelli@it.ibm.com

Threads

- Three (at least) different types
  - kernel threads
  - Lightweight processes
  - user threads

Giorgio Richelli
giorgio_richelli@it.ibm.com

Kernel Threads

- A kernel thread is responsible for executing a specific function.
- It shares the kernel text and global data, and has its own kernel stack.
- Independently scheduled.
- Useful to handle asynchronous I/O.
- Inexpensive
- Not entirely a new concept: pagedaemon nfsd

Giorgio Richelli
giorgio_richelli@it.ibm.com

Lightweight Processes

- LWP is a kernel-supported user thread.
- It belongs to a user process.
- Independently scheduled.
- Share the address space and other resources of the process.
- LWP should be synchronized on shared data.
- Blocking an LWP is expensive.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Lightweight processes

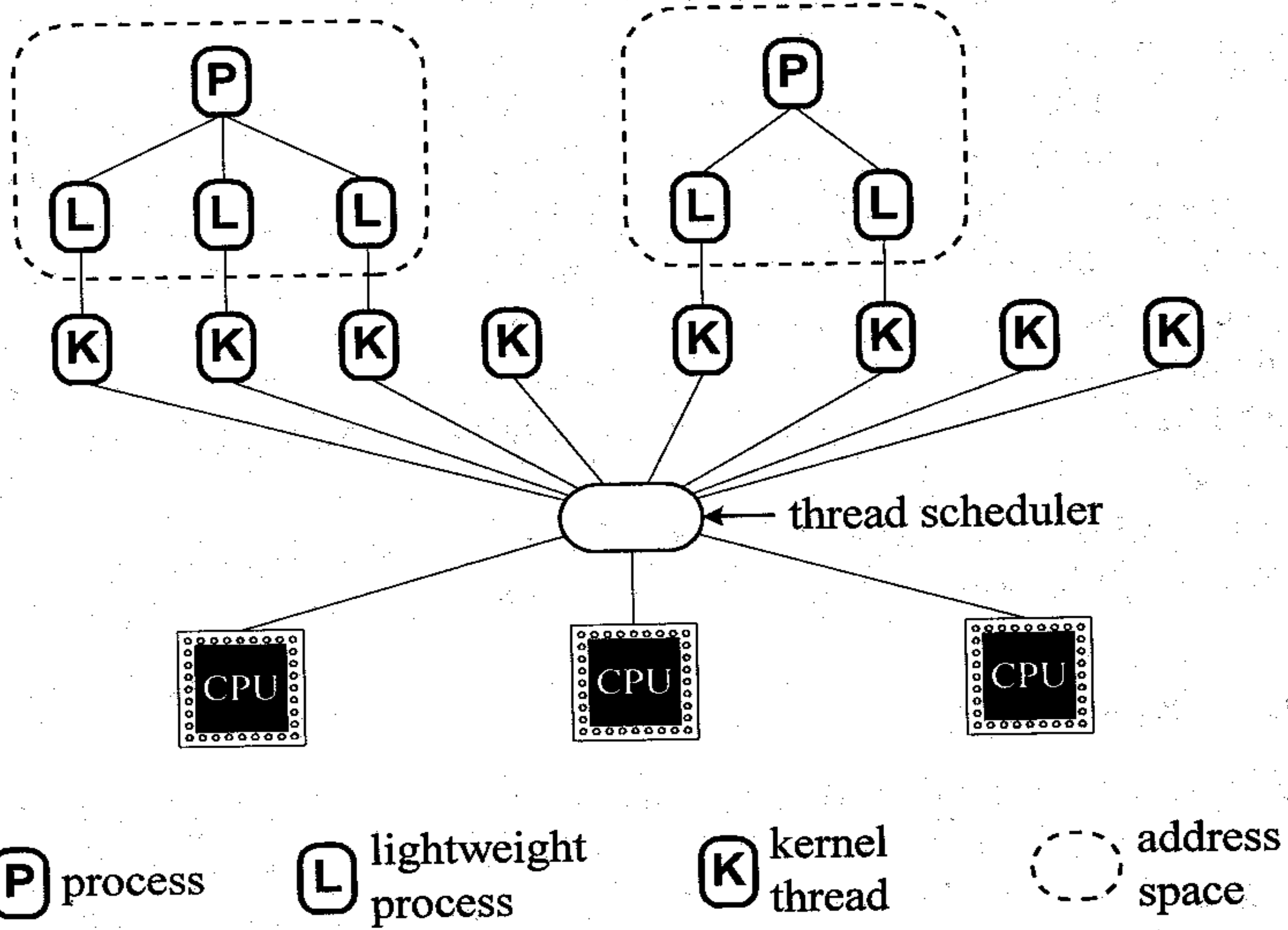


Figure 3-4. Lightweight processes.

Giorgio Richelli
giorgio_richelli@it.ibm.com

User Threads

- All thread management is done by the application
- The kernel is not aware of the existence of threads
- Thread switching does not require kernel mode privileges
- Scheduling is application specific

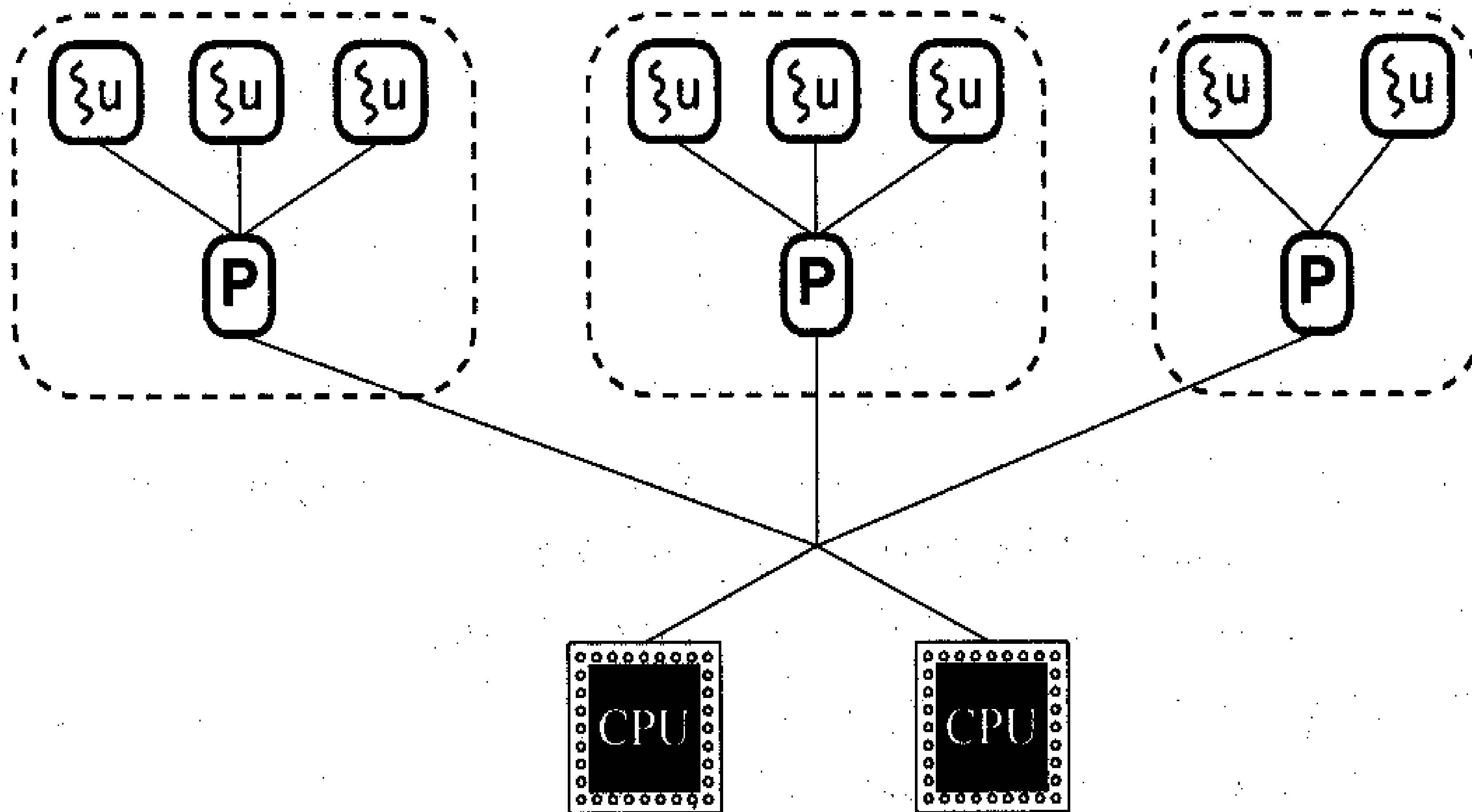
Giorgio Richelli
giorgio_richelli@it.ibm.com

User Threads

- Created by thread library such as *C-threads* (Mach) or *pthreads* (POSIX).
- A user-level library:
  - multiplex user threads on top of LWPs
  - provides facilities for inter-thread scheduling, context switching, and synchronization without involving the kernel.

Giorgio Richelli
giorgio_richelli@it.ibm.com

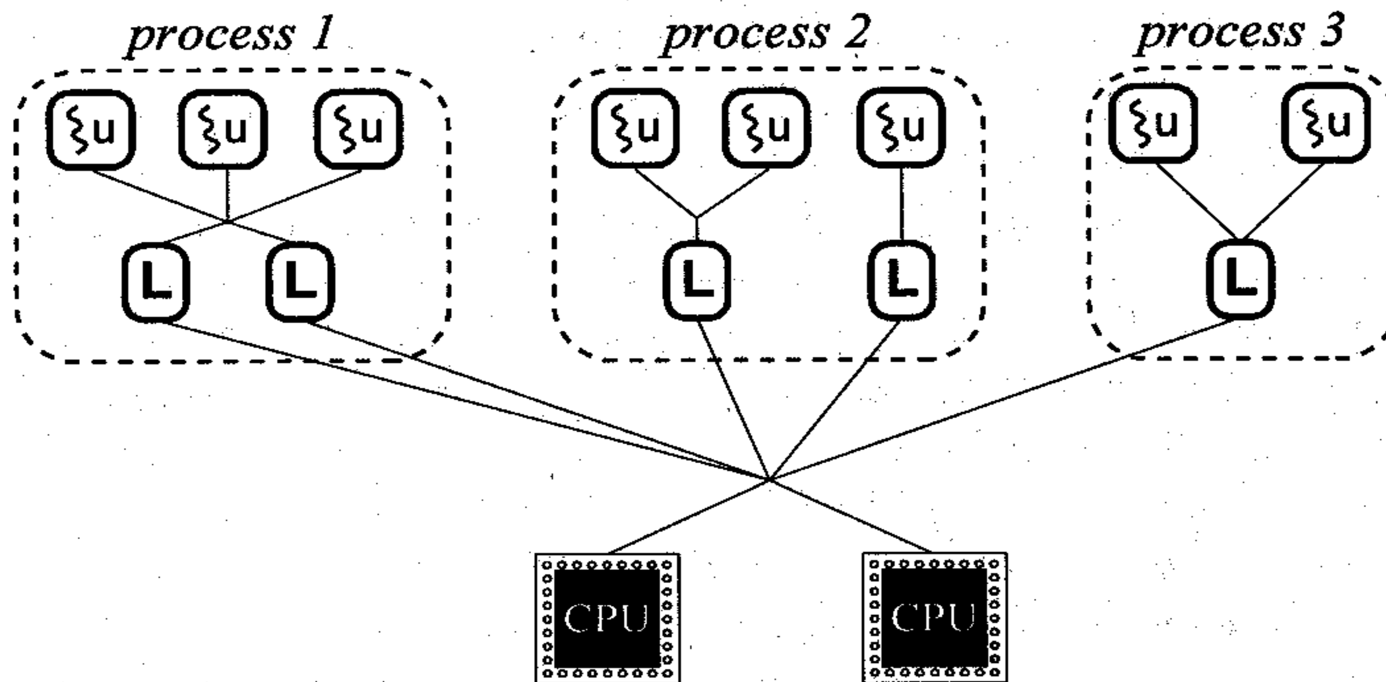
User Threads



(a) User threads on top of ordinary processes

Giorgio Richelli
giorgio_richelli@it.ibm.com

User Threads multiplexed



(b) User threads multiplexed on lightweight libraries

Figure 3-5. User thread implementations.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Thread Latencies

	Creation time (μs)	*Synchronization Time (μs)*
User thread	52	66
LWP	350	390
Process	1700	200

Giorgio Richelli
giorgio_richelli@it.ibm.com

Split scheduling

- The threads library schedules user threads
- The kernel schedules LWPs and processes.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Lightweight Process Design

- Semantics of fork
  - Create a child process.
  - Copy only the LWP into the new process (Posix)
  - Duplicate all the LWPs of the parent (Solaris).
- All LWPs share a set of file descriptor
- All LWPs share a common address space and may manipulate it concurrently through system calls such as *mmap* & *brk*.

Giorgio Richelli
giorgio_richelli@it.ibm.com

lseek Problems

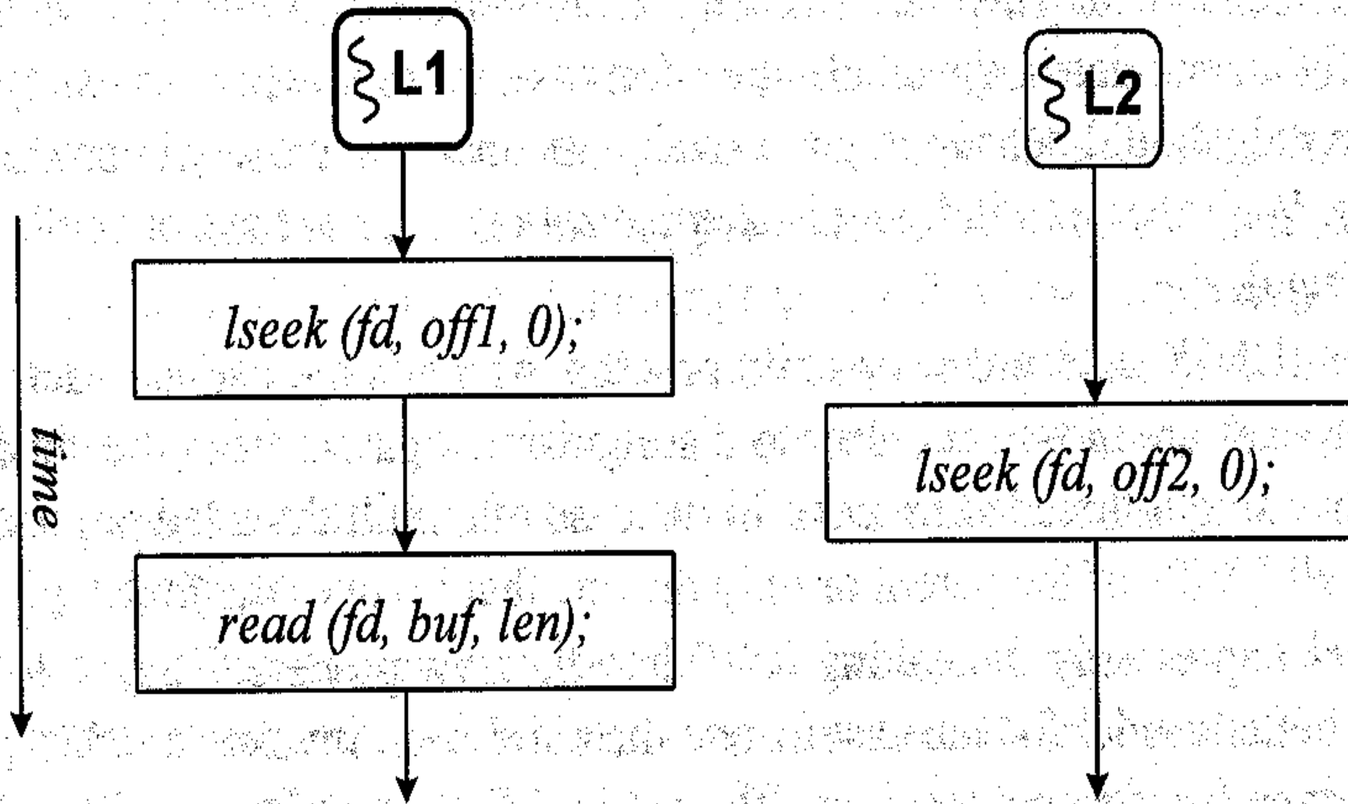


Figure 3-6. Problems with concurrent access to a file.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Signal Delivery and Handling

- Different methods:
  - Send the signal to each thread
  - Appoint a master thread in each process to receive all signals
  - Send the signal to any arbitrarily chosen thread
  - Use heuristics to determine the thread to which the signal applies
  - Create a new thread to handle each signal

Giorgio Richelli
giorgio_richelli@it.ibm.com

Stack Growth

- Overflows of stack: a segmentation violation fault.
- The kernel has no ideas about the user thread stack.
- It is the thread's responsibility to extend the stack or handle the overflows, the kernel responds by sending a SIGSEGV signal to the appropriate thread.

Giorgio Richelli
giorgio_richelli@it.ibm.com

User-Level Thread Libraries

- The API allows
  - Creating and terminating threads
  - Suspending and resuming threads
  - Assigning priorities to individual threads
  - Thread scheduling and context switching
  - Synchronizing activities through facilities such as semaphores and mutual exclusion locks
  - Sending messages from one thread to another
- The priority of a thread is simply a process-relative priority used by the threads scheduler to select a thread to run within the process.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Implementing Threads Libraries

- By LWPs:
  - Bind each thread to a different LWP.
  - Multiplex user threads on a (smaller) set of LWPs.
  - Allow a mixture of bound and unbound threads in the same process.

Giorgio Richelli
giorgio_richelli@it.ibm.com

User thread states

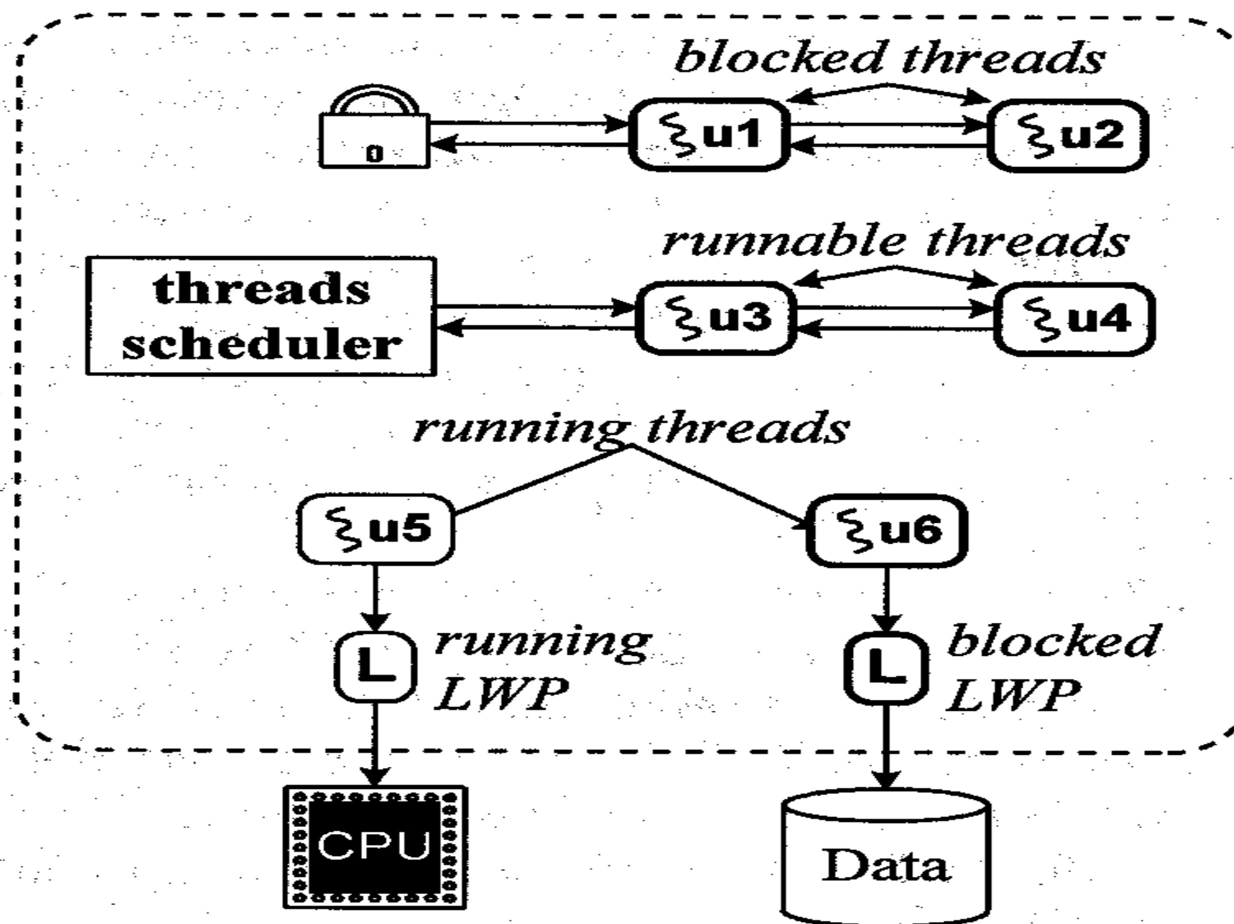


Figure 3-7. User thread states.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Linux: Processes or Threads?

- Linux uses a neutral term: tasks
- Traditional view
  - Threads exist "inside" processes
- Linux view
  - Threads: processes that share address space
  - Linux "threads" (tasks) are more like "LWPs"

Giorgio Richelli
giorgio_richelli@it.ibm.com

Thread Models

- Many-to-one
  - User-level threads; kernel doesn't know about them
- One-to-one
  - Linux standard model; each user-level thread corresponds to a kernel thread
- Many-to-many (m-to-n; m >= n)
  - Solaris, Next Generation POSIX Threads
  - Large number of user threads corresponds to a smaller number of kernel threads
  - More flexible; better CPU utilization

Giorgio Richelli
giorgio_richelli@it.ibm.com

clone()

- *fork()* is implemented as a wrapper around *clone()* with specific parameters
- __clone(fp, data, flags, stack)
  - "__" means "don’t call this directly"
  - fp is thread start function, data is params
  - flags is one or more of CLONE_ flags
  - stack is address of user stack
  - clone() calls do_fork() to do the work

Giorgio Richelli
giorgio_richelli@it.ibm.com

do_fork()

- Highlights
  - alloc_task_struct()
  - Copy current into new
  - find_empty_process()
  - get_pid()
  - Update ancestry
  - Copy components based on flags
  - copy_thread()
  - Link into task list, update nr_tasks
  - Set TASK_RUNNING
  - wake_up_process()

Giorgio Richelli
giorgio_richelli@it.ibm.com

Linux Kernel threads

- Linux has a small number of kernel threads that run continuously in the kernel (daemons)
  - No user address space (only kernel mapped)
- Creating: kernel_thread()
- Process 0: idle process
- Process 1
  - Spawns several kernel threads, before transitioning to user mode as */sbin/init*
  - kflushd (bdflush) – Flush dirty buffers to disk under "memory pressure"
  - kupdate – Periodically flushes old buffers to disk
  - kswapd – Swapping daemon
  - kpiod – No longer used in 2.4

Destroying Processes

- Termination
  - kill(), exit()
- Removal
  - wait()
- Terminating a process causes all threads to exit

Giorgio Richelli
giorgio_richelli@it.ibm.com

Cancelling a Thread

- A *cancel* is a mechanism by which a calling thread informs the specified thread to terminate as quickly as possible.
- Issuing a cancel does not guarantee that the canceled thread will receive or handle the cancel.
  - The canceled thread can delay processing the cancel after receiving it.
  - The calling thread can only rely on the fact that a cancel will eventually become pending in the designated thread (provided that the thread does not terminate beforehand).
  - Furthermore, the calling thread has no guarantee that a pending cancel will be delivered because delivery is controlled by the designated thread