First-class User Level Threads

- based on paper: “First-Class User Level Threads” by Marsh, Scott, LeBlanc, and Markatos
  - research paper, not merely an implementation report

User-level Threads

- Threads managed by a threads library
  - Kernel is unaware of presence of threads
- Advantages:
  - No kernel modifications needed to support threads
  - Efficient: creation/deletion/switches don't need system calls
  - Flexibility in scheduling: library can use different scheduling algorithms, can be application dependent
- Disadvantages
  - Need to avoid blocking system calls [all threads block]
  - Threads compete for one another
  - Does not take advantage of multiprocessors [no real parallelism]

User-level threads

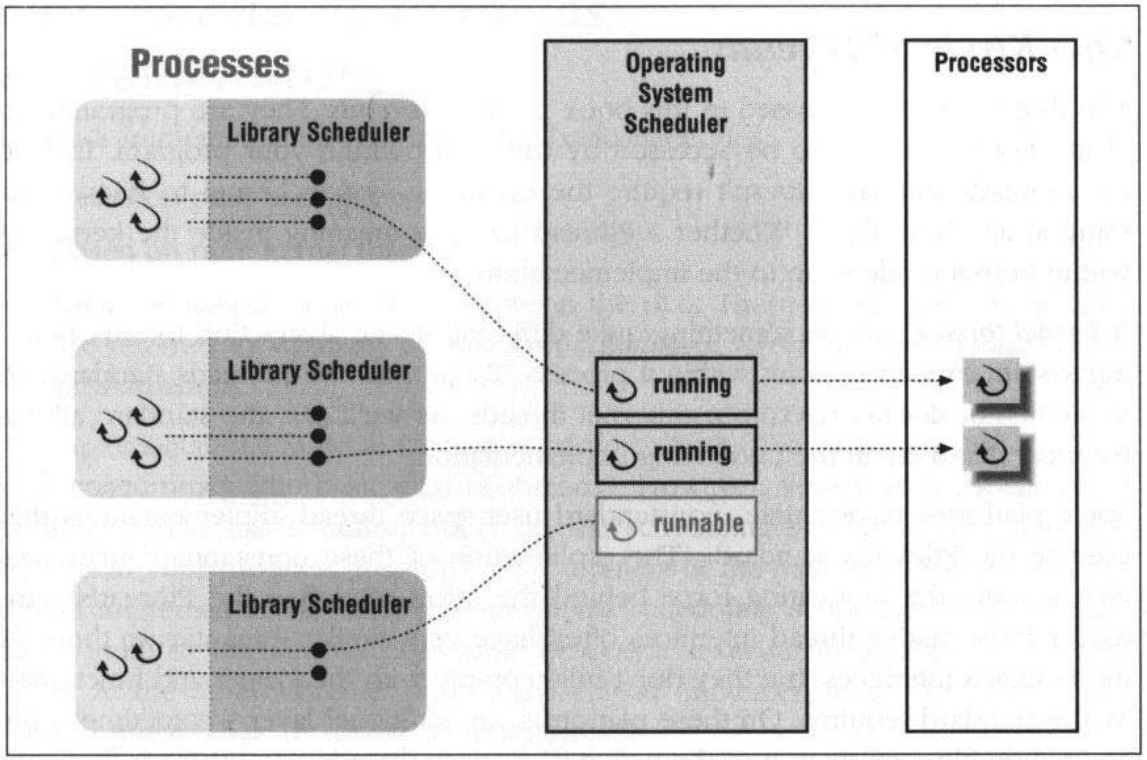

Figure 6–1: User-space thread implementations

Kernel-level threads

- Kernel aware of the presence of threads, but
  - Support 1000 u-l threads? – overhead on kernel resources
    - what if u-l thread doesn't need all info which kernel maintains per thread?
  - Kernel needs to provide general mechanisms
    - hard to tailor scheduling or synchronization operations

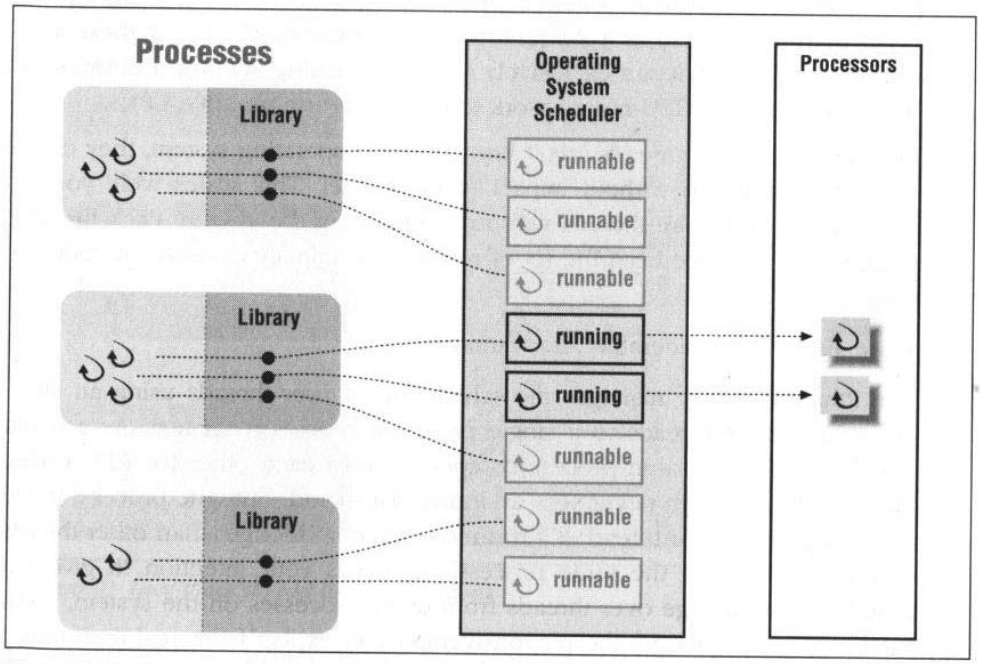

Figure 6–2: Kernel thread–based implementations

Lightweight Processes

- Several LWPs per heavy-weight process
- User-level threads package
  - Create/destroy threads and synchronization primitives
- Multithreaded applications – create multiple threads, assign threads to LWPs (one-one, many-one, many-many)
- Each LWP, when scheduled, searches for a runnable thread *[two-level scheduling]*
  - Shared thread table: no kernel support needed
- When a LWP thread block on system call, switch to kernel mode and OS context switches to another LWP

LWP Example

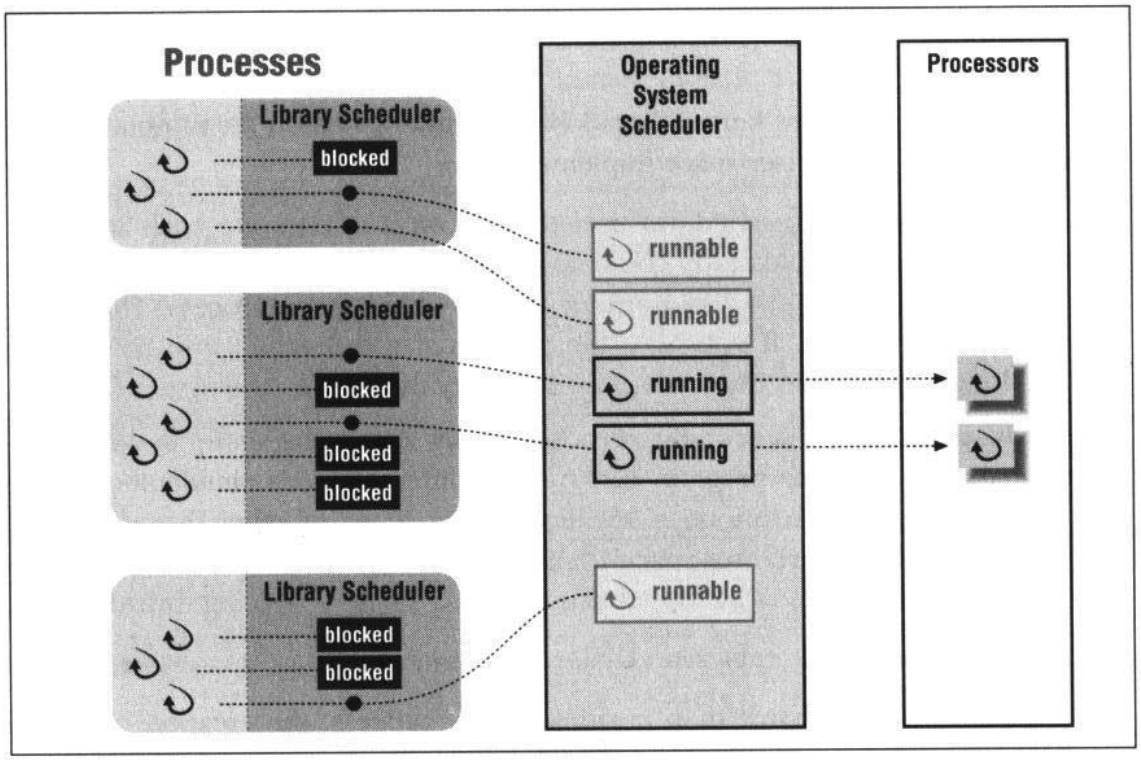

Figure 6–3: Two-level scheduler implementations

- lack of coordination between scheduling and synchronization persists
- blocking system calls will limit per-process resources
- cooperation among threads from different packages

Goal of paper

- present a minimal kernel interface for enabling the kernel to support user-level threads as if they were kernel-level threads

- work done in the context of Psyche: an experimental OS that attempts to integrate many models of parallelism that will co-exist and cooperate

User-level threads “second class”

- difficult or impossible to perform scheduling at appropriate times
  - many blocking system calls: e.g. read, write, oper, close, but also ioctl, mkdir, rmdir, rename, link, unlink
  - although some systems offer some non-blocking alternatives, (mutex)
- no coordination between scheduling and synchronization
  - a preempted thread may be holding a lock for which all other threads will wait
- no conventions for communication and synchronization across thread packages

Approach

- kernel thread interface is designed for use by user-level libraries
  - user level libraries do almost everything
    - create, destroy, synch, context switch….
  - kernel threads implement a virtual processor
  - kernel executed system calls and does coarse grain scheduling - preemtive scheduling
  - However -> kernel and user level library communicate by accessing information maintained by the other
    - e.g., blocking calls

Mechanisms

- shared kernel/user data
- software interrupts
  - maskable, vector specified by user (app)
- standard interface for schedulers

Shared kernel/user structures

- read/only access to kernel data
  - no system call needed to determine current processors number or process id
- user-writable data can be seen by the kernel
  - no system call needed to set up handlers for kernel events, such as timer expiration (valuable because it reduces the overhead of thread context switch)
  - pointer to software interrupt handler (i.e., library scheduler) and parameters
  - the kernel and library share a structure for the current thread
    - thread state (stack, registers, etc.)
- kernel and user would write to user-writable area in mutual exclusion by default

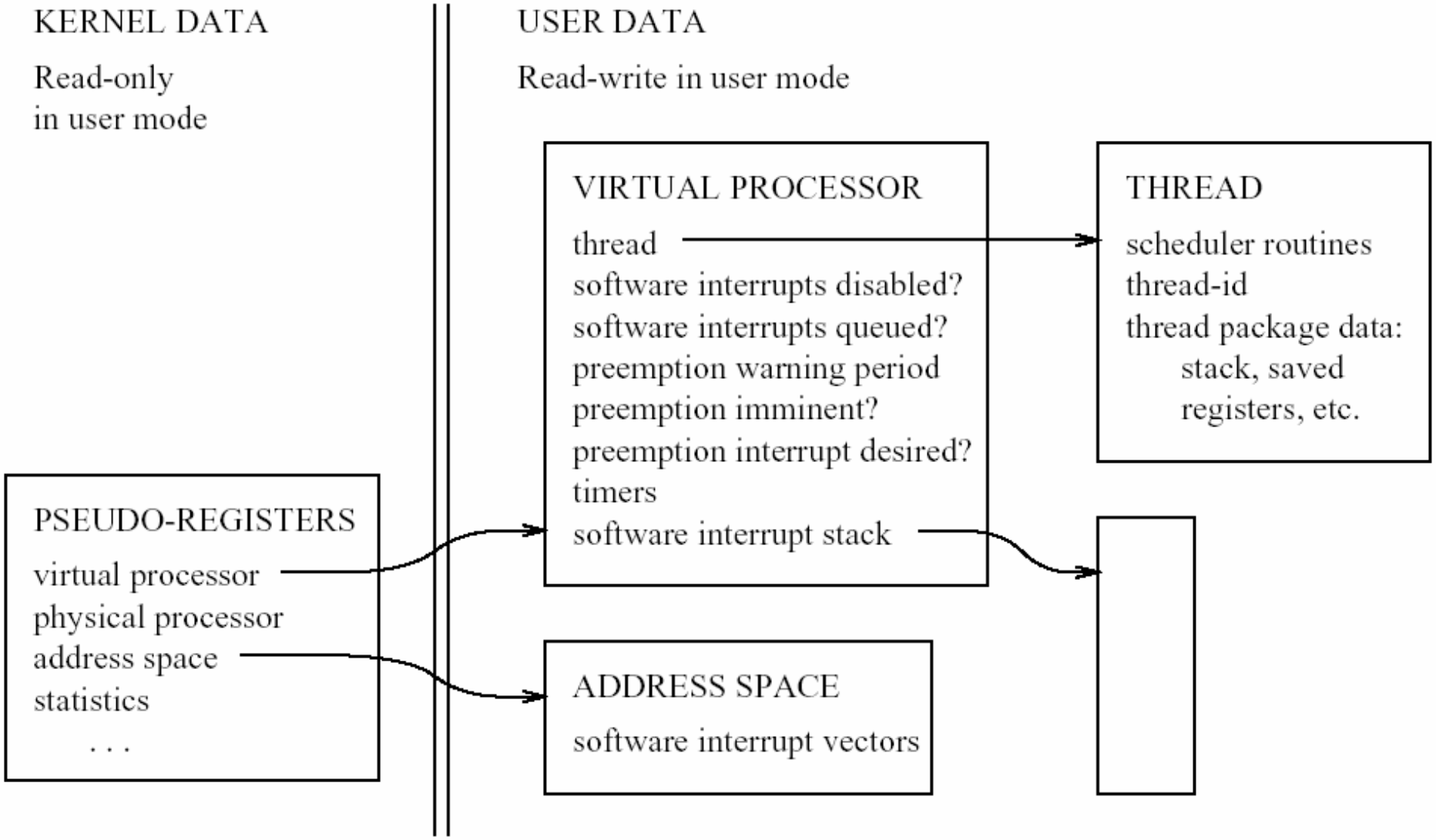

Figure 1: Shared Kernel/User Data Structures in Psyche

Software interrupts to user level library whenever scheduling may be required

- timer expiration (for time slicing!)
- imminent preemtion – *two minute warning* or *interrupt*
  - it is signaled when the thread is about to be preempted
  - can be used by threads to avoid acquiring spin locks when they are about to be descheduled
    - i.e., the implementation of spin locks checks for that; otherwise, newly scheduled threads will spend their time time slicing
    - useful because critical sections are usually short
  - good or bad?
- start and finish of blocking calls
  - all calls become non-blocking because the library scheduler gets to run before and after
  - kernel passes state of interrupted thread to the library (PC, stack pointer, etc.)

Standard interface for user-level schedulers (i.e., scheduling routines to block/unblock a thread)

- exported locations for functions implementing the interface
- hence, the threads from different packages can be blocked and unblocked
  - without needing to compile the program in a special language for a threads package
  - without needing to link the program with an implementation of the threads package
- the kernel does not call these functions directly, but they are there for other libraries to see
- e.g. atomic queue shared b/w thread packages A and B. when a thread adds data to the queue, it will need to find the routing to unblock threads waiting on the queue, possibly in another package

- paper advocates more communication between kernel and user space. is that what we want?
- general observations:
  - threads should not block in kernel unbeknown to the thread scheduler
  - kernel scheduling should be coordinated with user level scheduling