Threads
Chapter 4

Reading: 4.1,4.4, 4.5

Process Characteristics

- Unit of resource ownership - process is allocated:
  - a virtual address space to hold the process image
  - control of some resources (files, I/O devices...)
- Unit of dispatching - process is an execution path through one or more programs
  - execution may be interleaved with other process
  - the process has an execution state and a dispatching priority

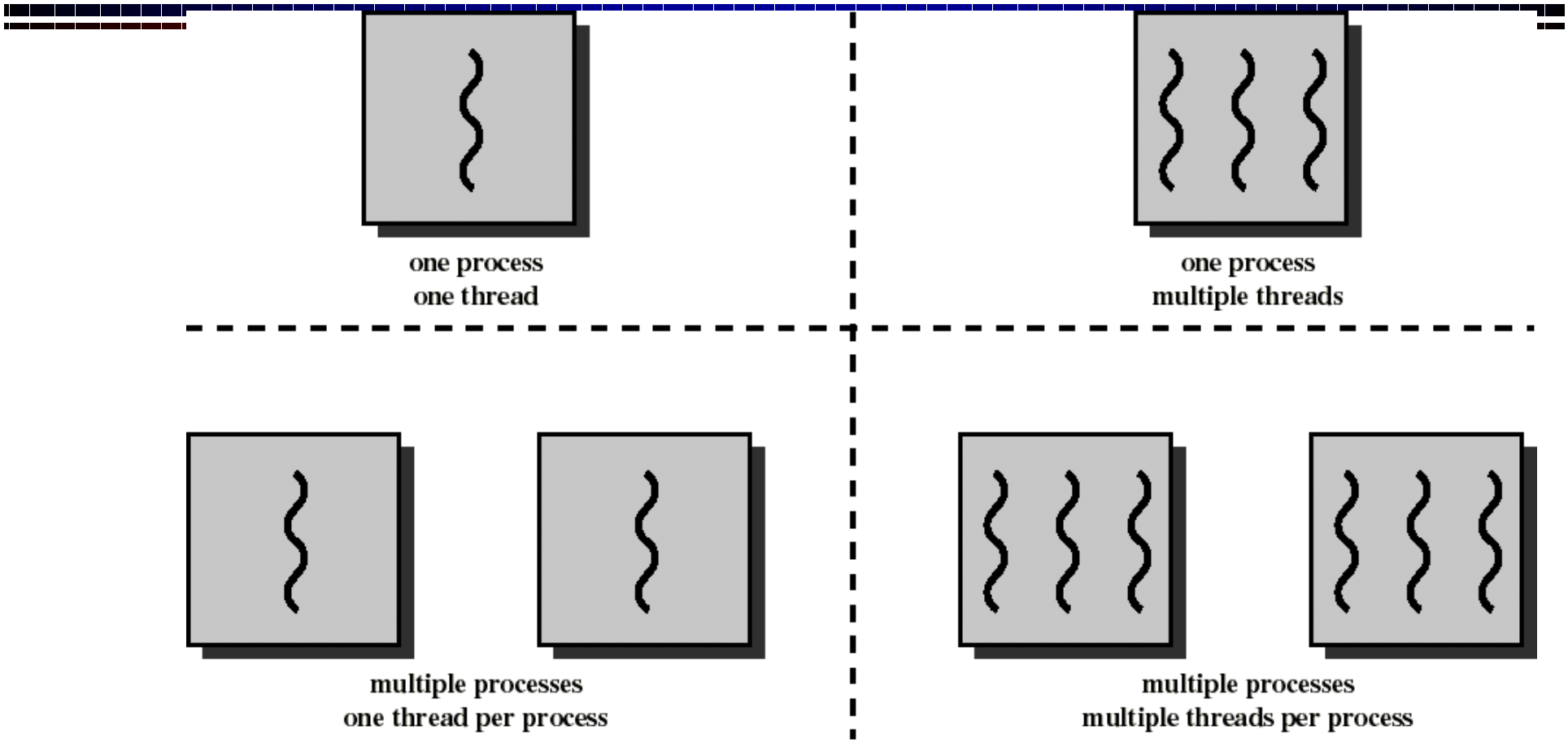
Process Characteristics

- These two characteristics are treated independently by some recent OS
- The unit of dispatching is usually referred to a thread or a lightweight process
- The unit of resource ownership is usually referred to as a process or task

Multithreading vs. Single threading

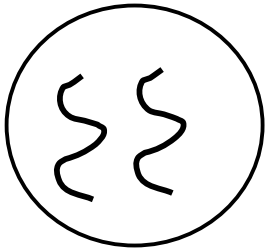
- Multithreading: when the OS supports multiple threads of execution within a single process
- Single threading: when the OS does not recognize the concept of thread
- MS-DOS support a single user process and a single thread
- UNIX supports multiple user processes but only supports one thread per process
- Solaris /NT supports multiple threads

Threads and Processes



Processes Vs Threads

- Have a virtual address space which holds the process image
  - Process: an address space, an execution context
  - Protected access to processors, other processes, files, and I/O resources
  - Context switch between processes expensive
- Threads of a process execute in a single address space
  - Global variables are shared
  - Context switch is less expensive (need only save thread state)

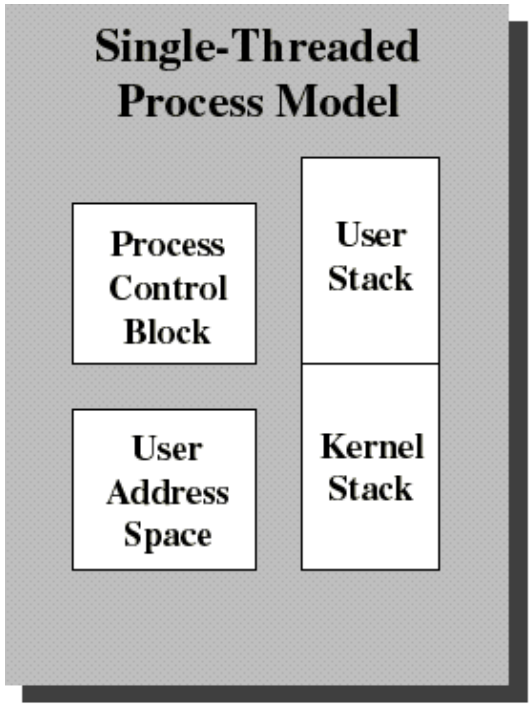


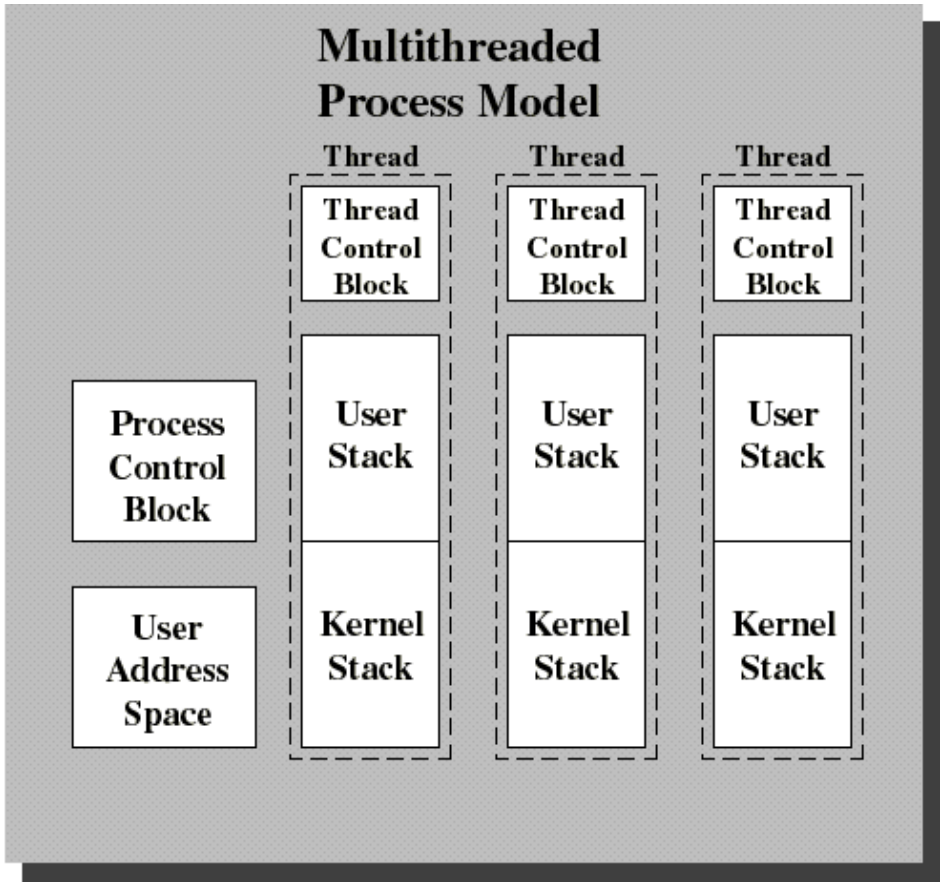
```
Class threadex{
Public static void main(String arg[]){
Int x=0;
My_thread  t1= new my_thread(x);
t1.start();
Thr_wait();
System.out.println(x)
}
}
Class my_thread extends Thread{
My_thread(int x){ this.x = x}
Public void run(){ x++;}
} private int x;}
```

Threads

- Has an execution state (running, ready, etc.)
- Saves thread context when not running
- Has an execution stack and some per-thread static storage for local variables
- Has access to the memory address space and resources of its process
  - all threads of a process share this
  - when one thread alters a (non-private) memory item, all other threads (of the process) sees that
  - a file open with one thread, is available to others

Single Threaded and Multithreaded Process Models





Thread Control Block contains a register image, thread priority and thread state information

Benefits of Threads vs Processes

- 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

Benefits of Threads

- Example: a file server on a LAN
- It needs to handle several file requests over a short period
- Hence more efficient to create (and destroy) a single thread for each request
- On a SMP machine: multiple threads can possibly be executing simultaneously on different processors
- Example2: one thread display menu and read user input while the other thread execute user commands

Application benefits of threads

- 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 an I/O, execution could possibly switch to another thread of the same application (instead of switching to another process)

Benefits of Threads

- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

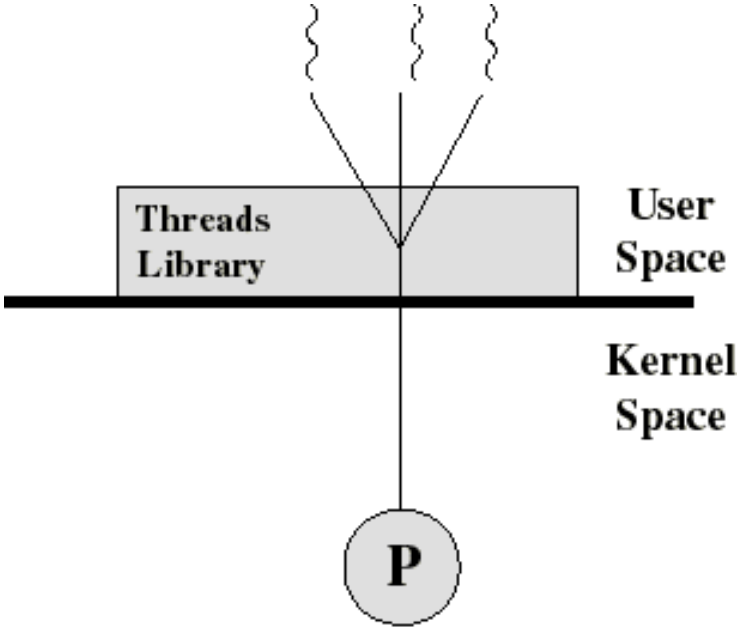
- Therefore necessary to synchronize the activities of various threads so that they do not obtain inconsistent views of the data (chap 5)

Threads States/package

- Three key states: running, ready, blocked
- Termination of a process, terminates all threads within the process
- Thread create, thread run
- Thread scheduling
  - Thread yield, specify priority, thread wait, thread sleep

User-Level Threads (ULT)

- The kernel is not aware of the existence of threads
- All thread management is done by the application by using a thread library
- Thread switching does not require kernel mode privileges (no mode switch)
- Scheduling is application specific



Threads library

- Contains code for:
  - creating and destroying threads
  - passing messages and data between threads
  - scheduling thread execution
  - saving and restoring thread contexts

Kernel activity for ULTs

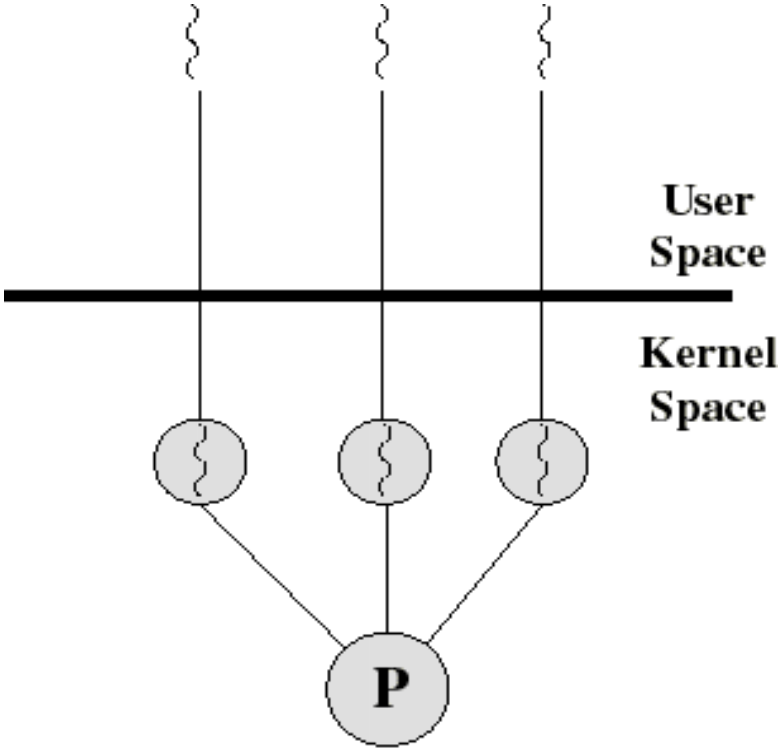
- The kernel is not aware of thread activity but it is still managing process activity
- When a thread makes a system call, the whole process will be blocked
- but for the thread library that thread is still in the running state
- So thread states are independent of process states

Advantages and inconveniences of ULT

- Advantages
  - Thread switching does not involve the kernel: no mode switching
  - Scheduling can be application specific: choose the best algorithm.
  - ULTs can run on any OS. Only needs a thread library
- Inconveniences
  - Most system calls are blocking and the kernel blocks processes. So all threads within the process will be blocked
  - The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors

Kernel-Level Threads (KLT)

- All thread management is done by kernel
- No thread library but an API to the kernel thread facility
- Kernel maintains context information for the process and the threads
- Switching between threads requires the kernel
- Scheduling on a thread basis
- Ex: Windows NT and OS/2

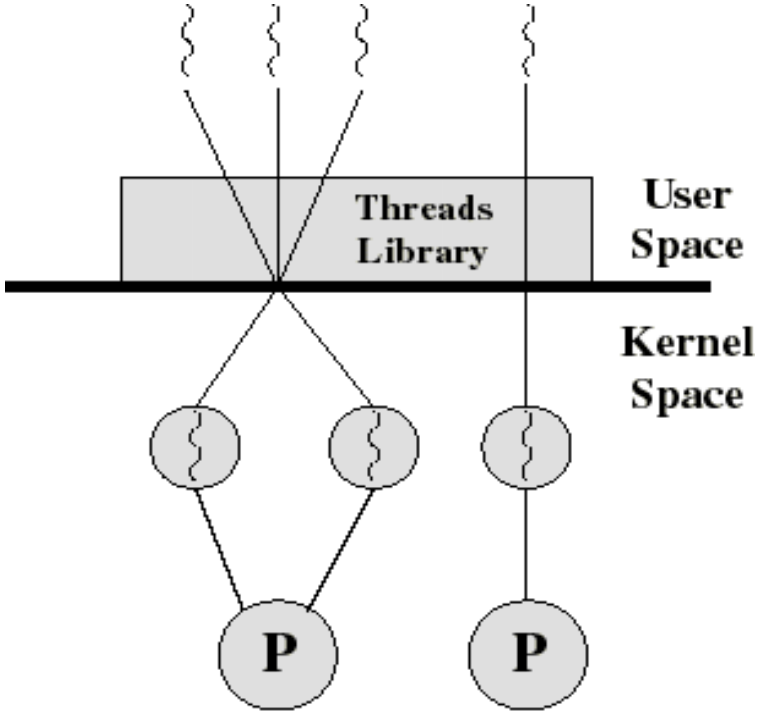


Advantages and inconveniences of KLT

- Advantages
  - the kernel can simultaneously schedule many threads of the same process on many processors
  - blocking is done on a thread level
  - kernel routines can be multithreaded
- Inconveniences
  - thread switching within the same process involves the kernel. We have 2 mode switches per thread switch
  - this results in a significant slow down

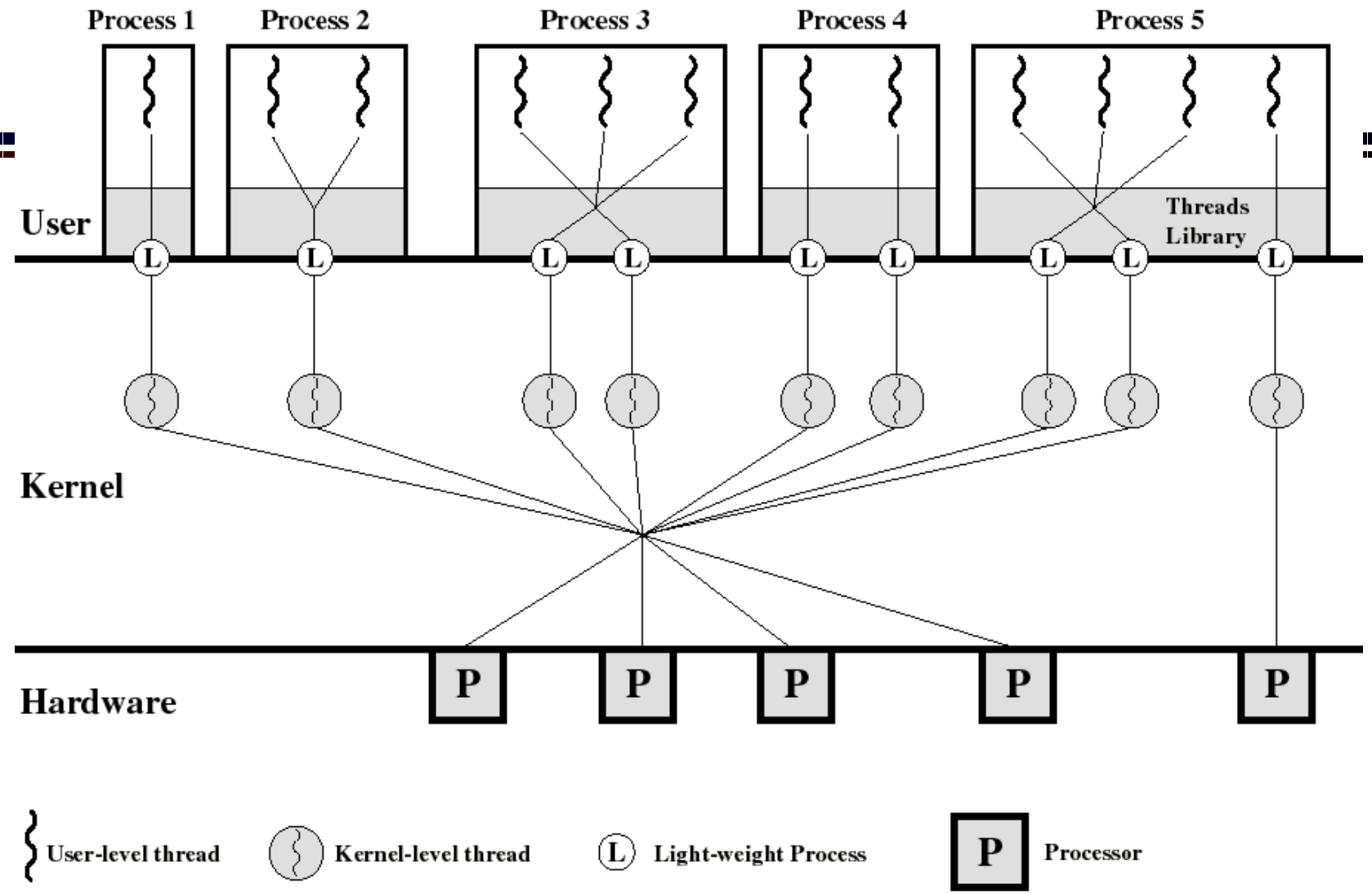
Combined ULT/KLT Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs
- May combine the best of both approaches
- Example is Solaris



Solaris

- Process includes the user's address space, stack, and process control block
- User-level threads (threads library)
  - invisible to the OS
  - are the interface for application parallelism
- Kernel threads
  - the unit that can be dispatched on a processor and it's structures are maintain by the kernel
- Lightweight processes (LWP)
  - each LWP supports one or more ULTs and maps to exactly one KLT
  - each LWP is visible to the application



Process 2 is equivalent to a pure ULT approach
Process 4 is equivalent to a pure KLT approach
We can specify a different degree of parallelism (process 3 and 5)

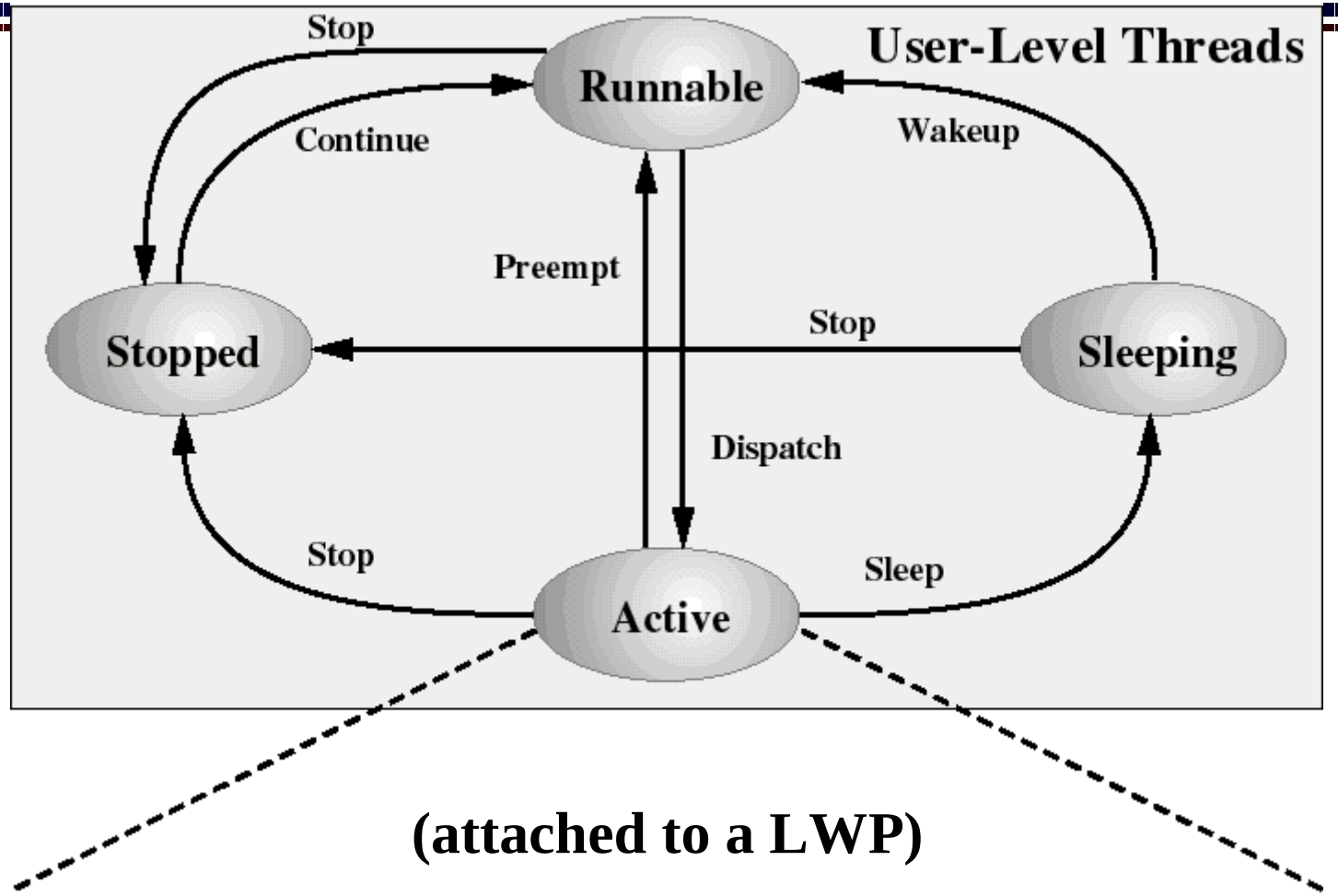
Solaris: versatility

- We can use ULTs when logical parallelism does not need to be supported by hardware parallelism (we save mode switching)
  - Ex: Multiple windows but only one is active at any one time
- If threads may block then we can specify two or more LWPs to avoid blocking the whole application

Solaris: user-level thread execution

- Transitions among these states is under the exclusive control of the application
  - a transition can occur only when a call is made to a function of the thread library
- It's only when a ULT is in the active state that it is attached to a LWP (so that it will run when the kernel level thread runs)
  - a thread may transfer to the sleeping state by invoking a synchronization primitive (chap 5) and later transfer to the runnable state when the event waited for occurs
  - A thread may force another thread to go to the stop state...

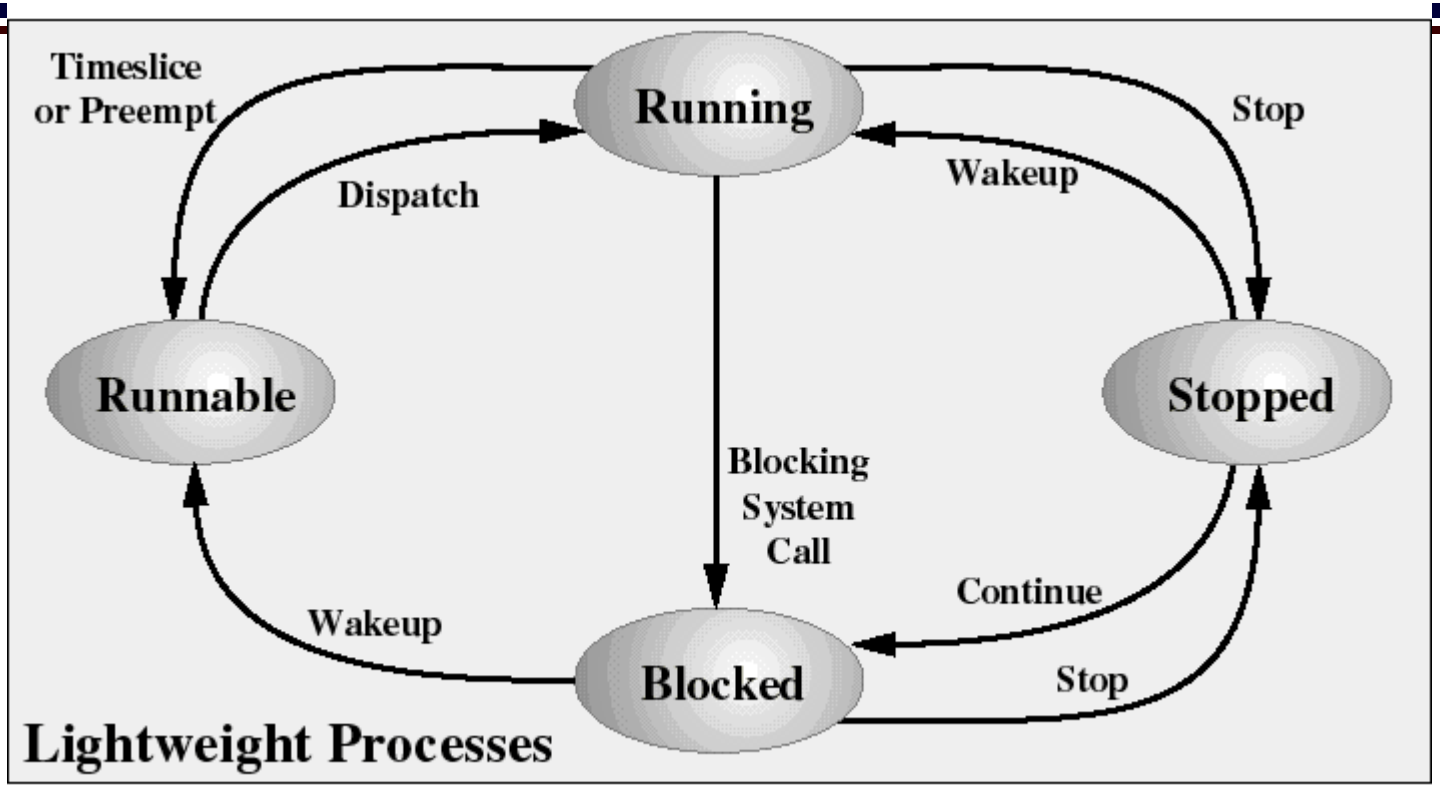
Solaris: user-level thread states



Decomposition of user-level *Active* state

- When a ULT is Active, it is associated to a LWP and, thus, to a KLT
- Transitions among the LWP states is under the exclusive control of the kernel
- A LWP can be in the following states:
  - running: when the KLT is executing
  - blocked: because the KLT issued a blocking system call (but the ULT remains bound to that LWP and remains active)
  - runnable: waiting to be dispatched to CPU

Solaris: Lightweight Process States



LWP states are independent of ULT states (except for bound ULTs)

Thread scheduling

- All runnable threads are on user level prioritized dispatch queue
- Priority can vary from 0 to infinity
- LWP picks the highest priority runnable thread from the dispatch queue
- When thread blocks at user level, then the LWP picks the next runnable thread