Process Management

Process Alternative - Threads

Lecture 5

Overview

- **Threads**
  - Motivation
  - Basic Idea
- **Threads models**
  - Kernel vs User Thread
  - Hybrid model
- **Thread in Unix**
  - POSIX thread:
    - Create, Exit, Synchronization
    - Exploration

Motivation for Thread

- Process is expensive:
  - Process creation under the `fork()` model:
    - Duplicate memory space
    - Duplicate most of the process context etc.
  - Context switch:
    - Requires saving/restoration of process Information

- It is hard for independent processes to communicate with each other:
  - Independent memory space ➔ No easy way to pass information
  - Requires Inter-Process Communication (IPC)

Motivation for Thread (cont)

- Thread was invented to overcome the problems with process model
  - Started out as a "quick hack" and eventually matured to be very popular mechanism

- **Basic Idea:**
  - A traditional process has a single thread of control:
    - Only one instruction of the whole program is executing at any time
  - We "simply" add more threads of control to the same process:
    - Multiple parts of the programs is executing at the same time conceptually

Threads of control: Illustration

- Suppose we are preparing lunch, which consists of the following tasks:
  - Steam rice
  - Fry fish
  - Cook soup
- A pseudo-C program:

```
int main()
{
    steamRice( twoBowls );
    fryFish( bigFish );
    cookSoup( cornSoup );

    printf( "Lunch READY!!\n" );
    return 0;
}
```

Threads of control: Illustration (cont)

- Process with a single thread will go through the functions sequentially

```
int main()
{
    steamRice( twoBowls );
    fryFish( bigFish );
    cookSoup( cornSoup );

    printf( "Lunch Ready\n" );
    return 0;
}
```

Single PC

Time

- Suppose the tasks are independent:
  - Let's try to have multiple threads of control

Multiple "threads of control" with **fork()**

```
int result;
result = fork();        PC 1
if (result != 0){
    result = fork();
    if (result != 0){
        steamRice( twoBowls );
    } else {
        fryFish( bigFish );
     }
} else {
    cookSoup( cornSoup );
}
...  ...
```

```
int result;
result = fork();        PC 2
if (result != 0){
    result = fork();
    if (result != 0){
        steamRice( twoBowls );
    } else {
        fryFish( bigFish );
     }
} else {
    cookSoup( cornSoup );
}
...  ...
```

We effectively have two "threads" of control after fork()

New approach: Multi-Threading

```
int main()
{
    Thread 1 do {  steamRice( twoBowls ); }

    Thread 2 do { fryFish( bigFish ); }

    Thread 3 do { cookSoup( cornSoup ); }

    Wait for all threads to finish;

    printf( "Lunch Ready\n");
    return 0;
}
```

PC 1

PC 2

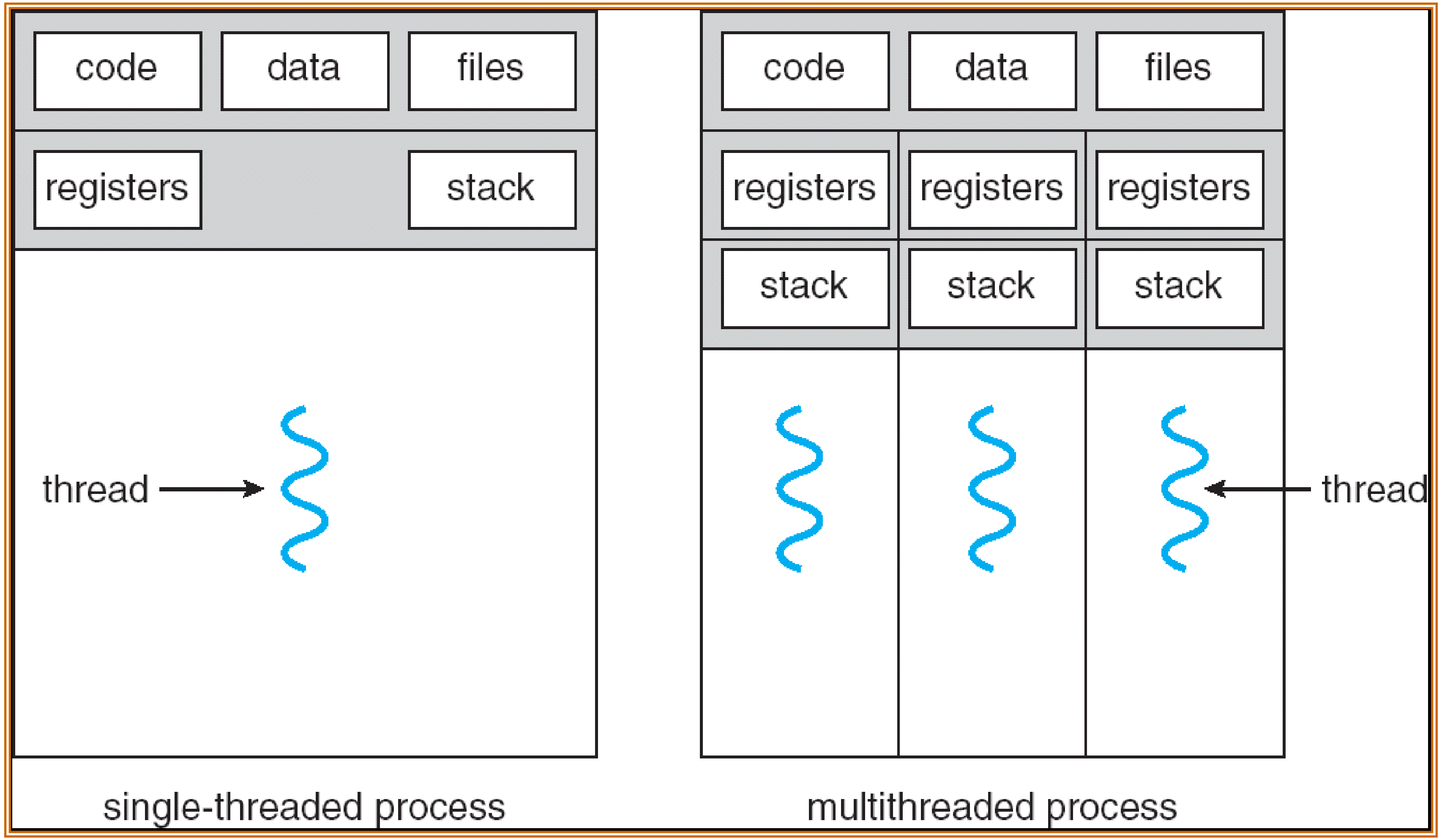
PC 3

- Note:
  - Pseudocode ☺
  - The three threads are concurrent

Process and Thread

- A single process can have multiple threads:
  - known as **multithreaded process**
- Threads in the same process shares:
  - **Memory Context**: Text, Data, Heap
  - **OS Context**: Process id, other resources like files, etc.
- Unique information needed for each thread:
  - Identification (usually **thread id**)
  - Registers (General purpose and special)
  - "Stack" (more about this later)

Process and thread: Illustration



- Taken from Operating System Concepts (7th Edition) by Silberschatz, Galvin & Gagne, published by Wiley

Process Context Switch VS Thread Switch

- Process context switch involves:
  - OS Context
  - Hardware Context
  - Memory Context

- Thread switch within the same process involves:
  - Hardware context:
    - Registers
    - "Stack" (actually just changing FP and SP registers)

- Thread is much “lighter” than process:
  - a.k.a. ***lightweight process***

Threads: **Benefits**

Economy:

- Multiple threads in the same process requires much less resources to manage compared to multiple processes

Resource sharing:

- Threads share most of the resources of a process
- No need for additional mechanism for passing information around

Responsiveness:

- Multithreaded programs can appear much more responsive

Scalability:

- Multithreaded program can take advantage of multiple CPUs

Thread: Problems

- **System Call Concurrency**
  - Parallel execution of multiple threads → parallel system call possible
    - Have to guarantee correctness and determine the correct behavior

- **Process Behavior:**
  - Impact on process operations
  - Example:
    - **`fork()`** duplicate process, how about threads?
    - If a single thread executes **`exit()`**, how about the whole process?
    - If a single thread calls **`exec()`**, how about other threads?

Thread Models

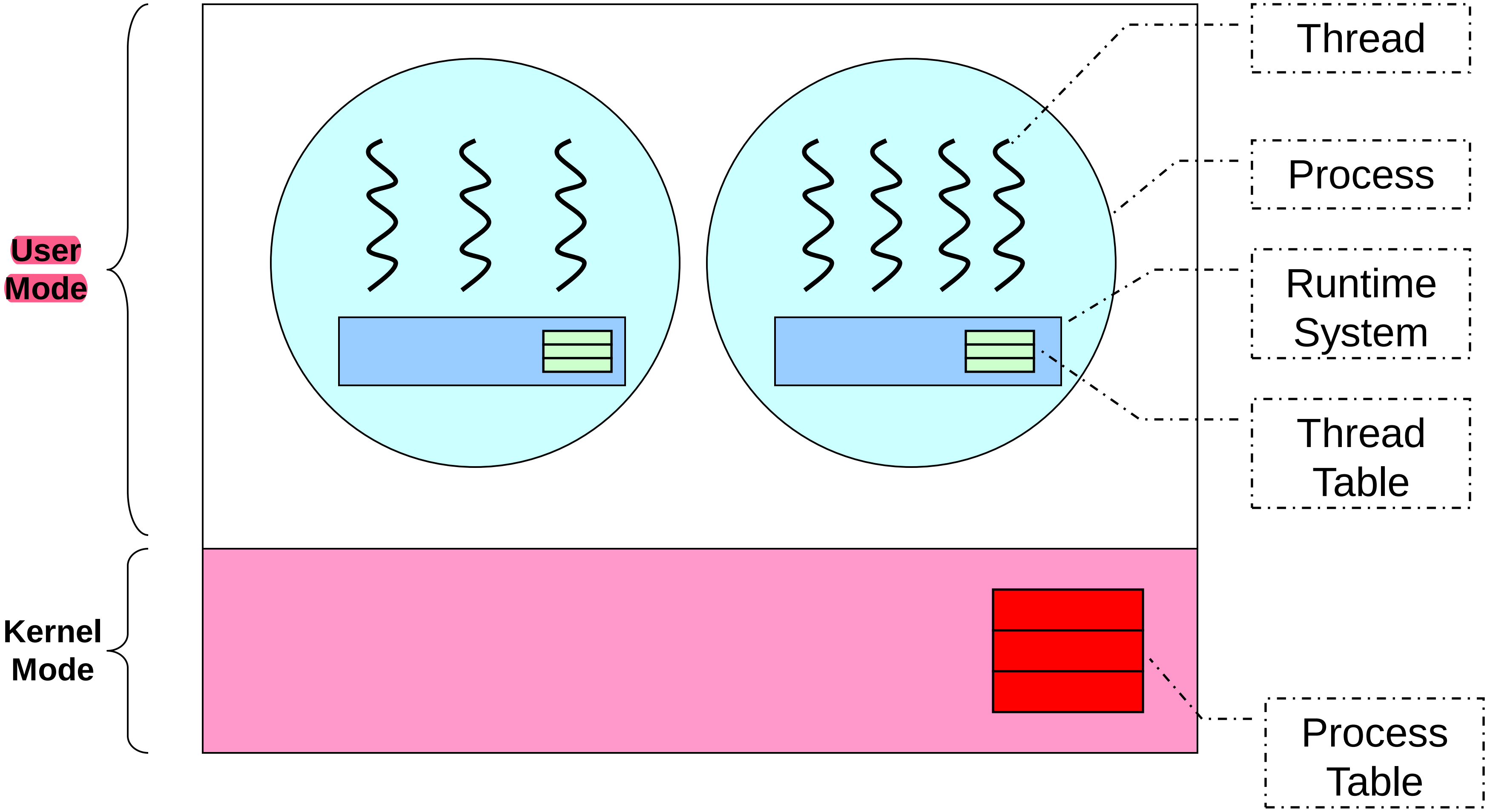
What are the ways to support threads?

Two Major Thread Implementations

- **User Thread:**
  - Thread is implemented as a **user library**
    - A runtime system (in the process) will handle thread related operation
  - Kernel is **not aware** of the threads in the process

- **Kernel Thread:**
  - Thread is implemented in the OS
    - Thread operation is handled as system calls
  - Thread-level scheduling is possible:
    - Kernel schedule by threads, instead of by process
  - Kernel may make use of threads for its own execution

User Thread: Illustration



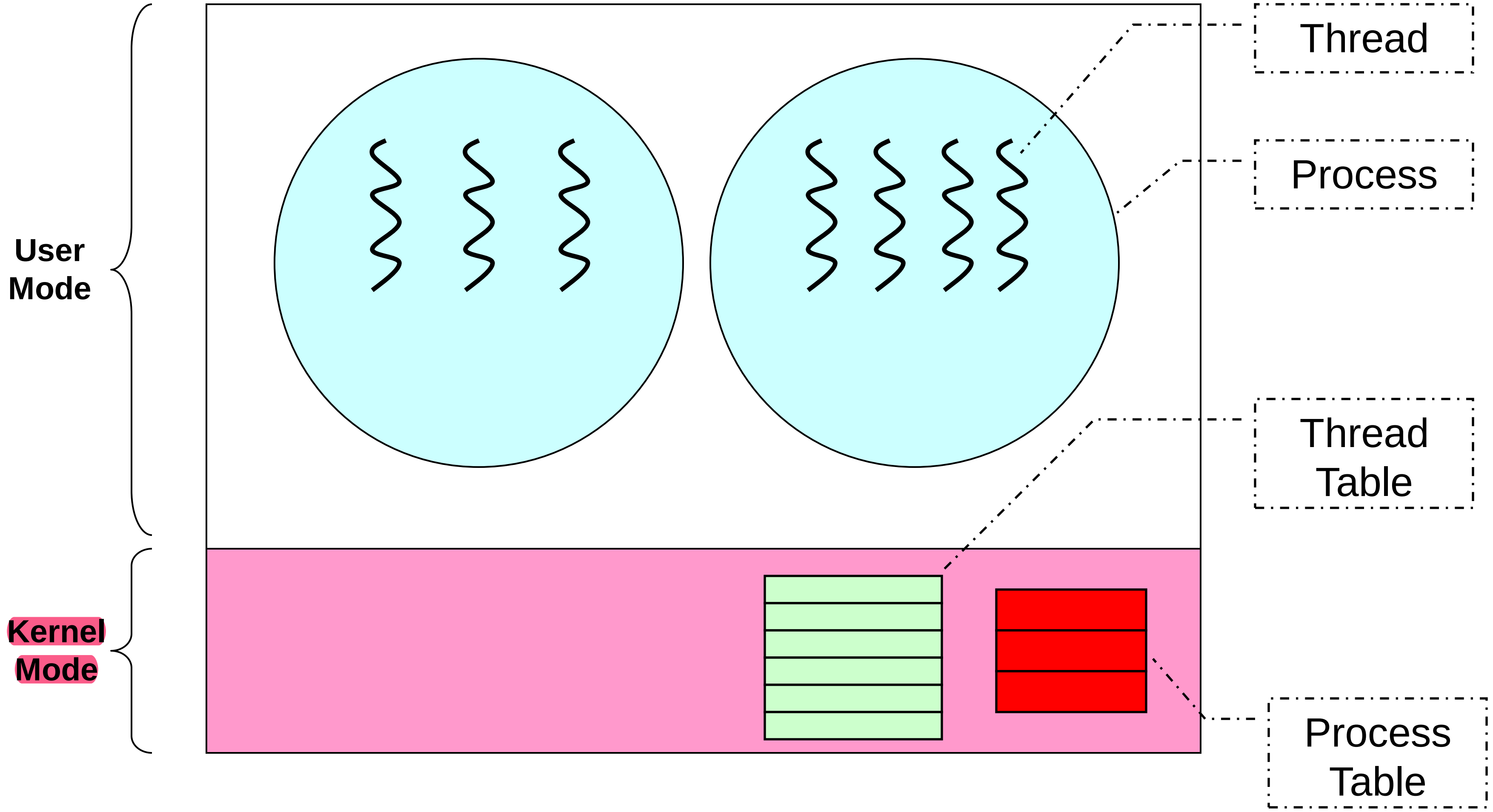
- Adapted from Modern Operating System Concepts (3rd Edition) by Andrew Tanenbaum, published by Pearson

User Thread: Pros and Cons

- **Advantages:**
  - Can have multithreaded program on ANY OS
  - Thread operations are just library calls
  - Generally more configurable and flexible
    - e.g., customized thread scheduling policy

- **Disadvantages:**
  - OS is not aware of threads, scheduling is performed at process level
    - One thread blocked → Process blocked → All threads blocked
    - Cannot exploit multiple CPUs!

Kernel Thread: Illustration



- Adapted from Modern Operating System Concepts (3rd Edition) by Andrew Tanenbaum, published by Pearson

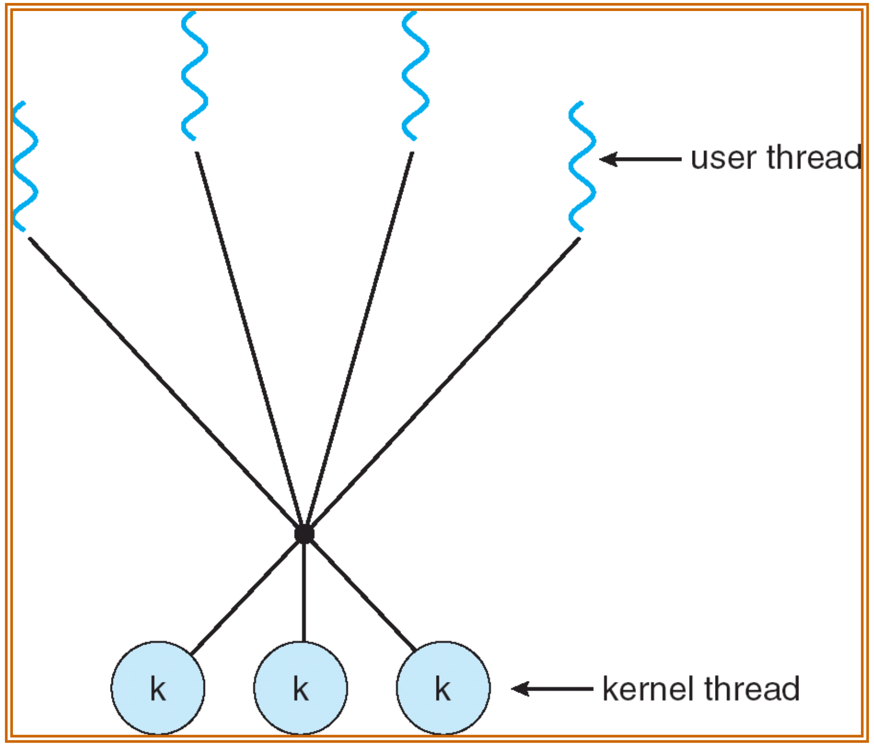
Kernel Threads: Pros and Cons

- **Advantages:**
  - Kernel can schedule on thread levels:
    - More than 1 thread in the same process can run simultaneously on multiple CPUs
- **Disadvantages:**
  - Thread operations is now a system call!
    - Slower and more resource intensive
  - Generally less flexible:
    - Used by all multithreaded programs
    - If implemented with many features:
      - Expensive, overkill for simple program
    - If implemented with few features:
      - Not flexible enough for some programs

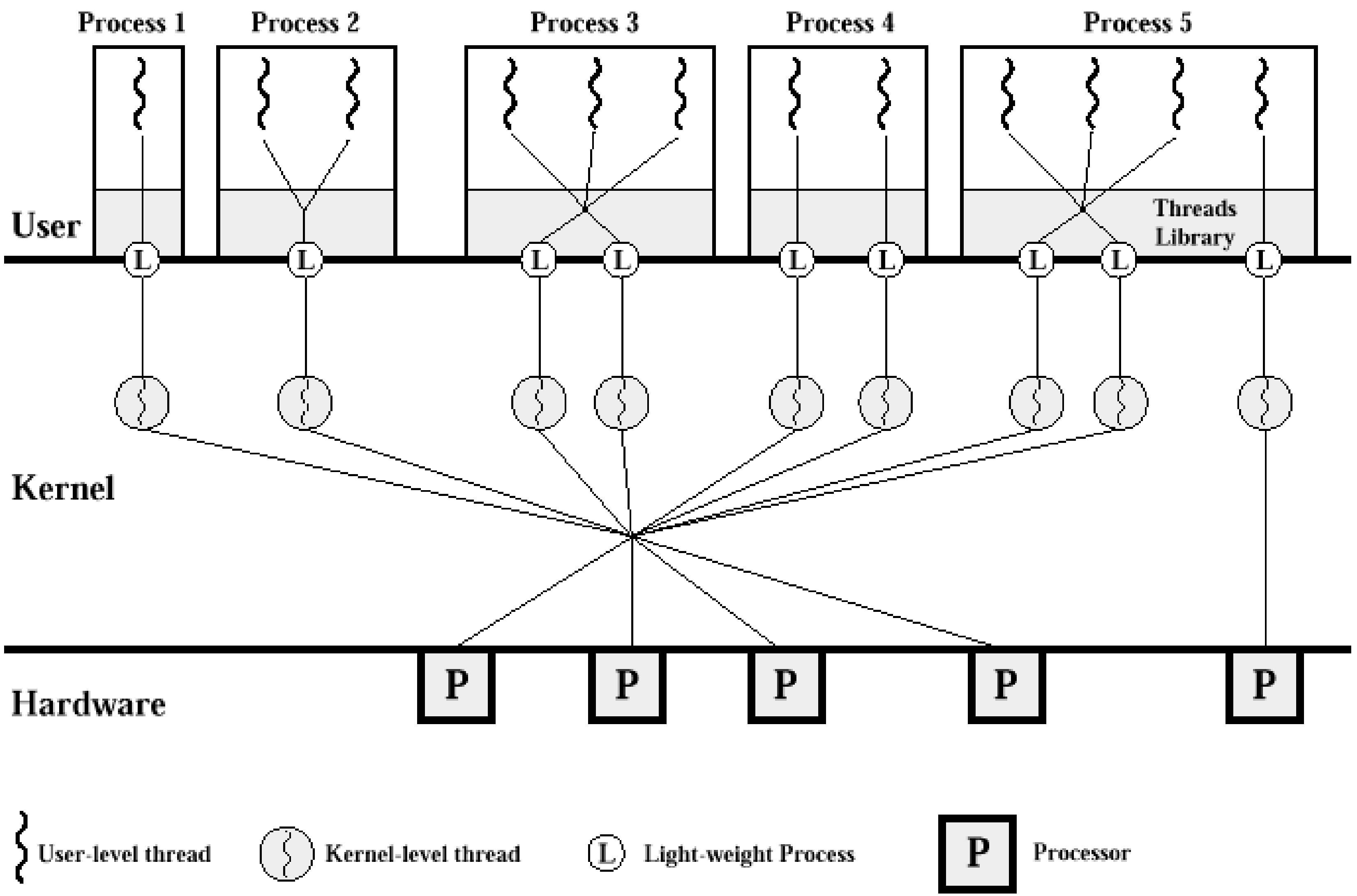
Hybrid Thread Model

- Have both Kernel and User threads
  - OS schedule on kernel threads only
  - User thread can bind to a kernel thread
- Offer great flexibility
  - Can limit the concurrency

  of any process / user



Hybrid Model Example: **Solaris**



Threads on Modern Processor

- Threads started off as a software mechanism
  - User space library ➔ OS aware mechanism

- There are hardware support on modern processors:
  - Essentially supply multiple sets of registers (GPRs, and special registers) to allow threads to run natively and in parallel on the same core
  - known as **simultaneous multi-threading (SMT)**
    - Example: **Hyperthreading** on Intel processor

POSIX Threads

A popular thread API

POSIX Threads: **pthread**

- Standard defined by the IEEE
  - Supported by most Unix variants

- Defines the API as well as the behavior
  - However, **implementation** is not specified
  - So, ***pthread*** can be implemented as user / kernel thread

- Will show a few example to highlight the differences between process and thread only

Basics of **pthread**

- Header File:
  - `#include <pthread.h>`

- Compilation (flag is system dependent):
  - `gcc XXXX.c -lpthread`

- Useful datatypes:
  - **`pthread_t`** : Data type to represent a thread id (TID)
  - **`pthread_attr`**: Data type to represents attributes of a thread

pthread Creation Syntax

```
int pthread_create(
        pthread_t* tidCreated,
        const pthread_attr_t* threadAttributes,
        void* (*startRoutine) (void*),
        void* argForStartRoutine  );
```

- Returns (0 = success; !0 = errors)

- Parameters:
  - `tidCreated`: Thread Id for the created thread
  - `threadAttributes`: Control the behavior of the new thread
  - `startRoutine`: Function pointer to the function to be executed by thread
  - `argForStartRoutine`: Arguments for the `startRoutine` function

pthread Termination Syntax

```
int pthread_exit( void* exitValue );
```

- Parameters:
  - **exitValue:** Value to be returned to whoever synchronize with this thread (more later)

- If ***pthread_exit***() is not used, a *pthread* will terminate automatically at the end of the **startRoutine**
  - If a “return XYZ;” statement is used, then “XYZ” is captured as the exitValue
  - Otherwise, the exitValue is not well defined

pthread Creation & Termination: Example

```
//header files not shown

void* sayHello( void* arg )
{
     printf("Just to say hello!\n");
     pthread_exit( NULL );
}

int main()
{
    pthread_t tid;

    pthread_create( &tid, NULL, sayHello, NULL );
    printf("Thread created with tid %i\n", tid);

    return 0;
}
```

Function to be executed by a pthread

Pthread Termination

Pthread Creation

pthread: Sharing of memory space

```
//header files not shown
int globalVar;        Variable shared between pthreads

void* doSum( void* arg)
{     int i;
      for (i = 0; i < 1000; i++)
           globalVar++;
}                                 Using a shared variables

int main()
    pthread_t tid[5];     //5 threads id
    int i;
    for (i = 0; i < 5; i++)
        pthread_create( &tid[i], NULL, doSum, NULL );
    printf("Global variable is %i\n", globalVar);
    return 0;
}
```

- What is the sum that we get (or should get)?

pthread Simple Synchronization - Join

```
int pthread_join( pthread_t threadID,
                  void **status );
```

- To wait for the termination of another ***pthread***:
- Returns(0 = success; !0 = errors)
- Parameters:
  - `threadID`: `TID` of the ***pthread*** to wait for
  - `status`: Exit value returned by the target ***pthread***

Pthread: Sharing of memory space V2.0

```
//header files not shown
int globalVar;

void* doSum( void* arg)
{ //same as before }

int main()
    pthread_t tid[5];     //5 threads id
    int i;
    for (i = 0; i < 5; i++)
        pthread_create( &tid[i], NULL, doSum, NULL );

    //Wait for all threads to finish
    for (i = 0; i < 5; i++)
        pthread_join( tid[i], NULL );

    printf("Global variable is %i\n", globalVar);
    return 0;
}
```

Pthread Synchronization

Pthread: A lot more!

- There are more interesting stuff about ***pthread***:
  - Yielding (giving up CPU voluntarily)
  - Advanced synchronization
  - Scheduling policies
  - Binding to kernel threads
  - Etc.

- As we cover new topics, you can explore the ***pthread*** library to see the application!

Summary

- Thread:
  - Motivation
  - Difference between thread and process

- Thread Models:
  - User vs Kernel thread

- Simple introduction to ***pthread*** library

Reference

- Modern Operating System (3rd Edition)
  - Chapter 2.2
- Operating System Concepts (7th Edition)
  - Chapter 4
- Linux Pthread Man Pages
  - "man pthread……" for more