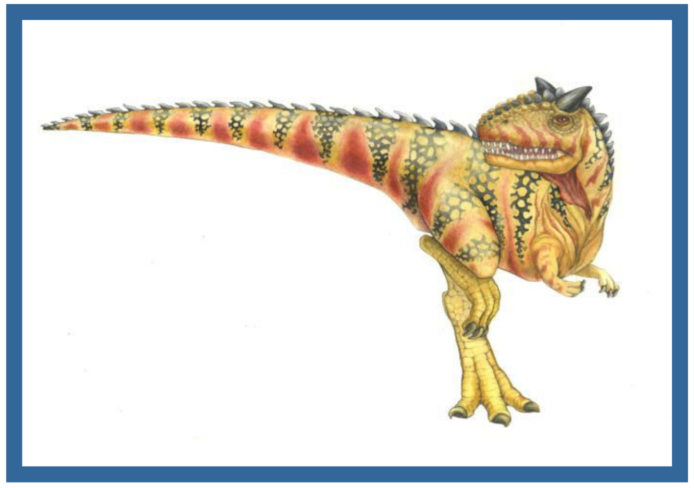
Chapter 4: Multithreaded Programming



Chapter 4: Threads

- Concept of threads.
- Why multithreading?
- Thread API for Linux.
- Some programming examples.

Objectives

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
- To discuss the API for Linux threads.

Motivation

- We have seen in chapter 3 that a process is a “unit of work” in operating systems.
- It is a program that is executing a sequence of instructions.
- While executing, a process needs access to resources such as: CPU, memory, I/O devices, files.
- **Key idea of threads: it is possible to have several such separate sequences of instruction execution <u>within</u> a process.**
- Each such sequence (of instructions) is called a “thread”.
- It may be possible to examine the code of a process and identify parts that can potentially be run independently of each other.
  - In other words, in parallel.
- All threads operate in the same address space of the creator process.

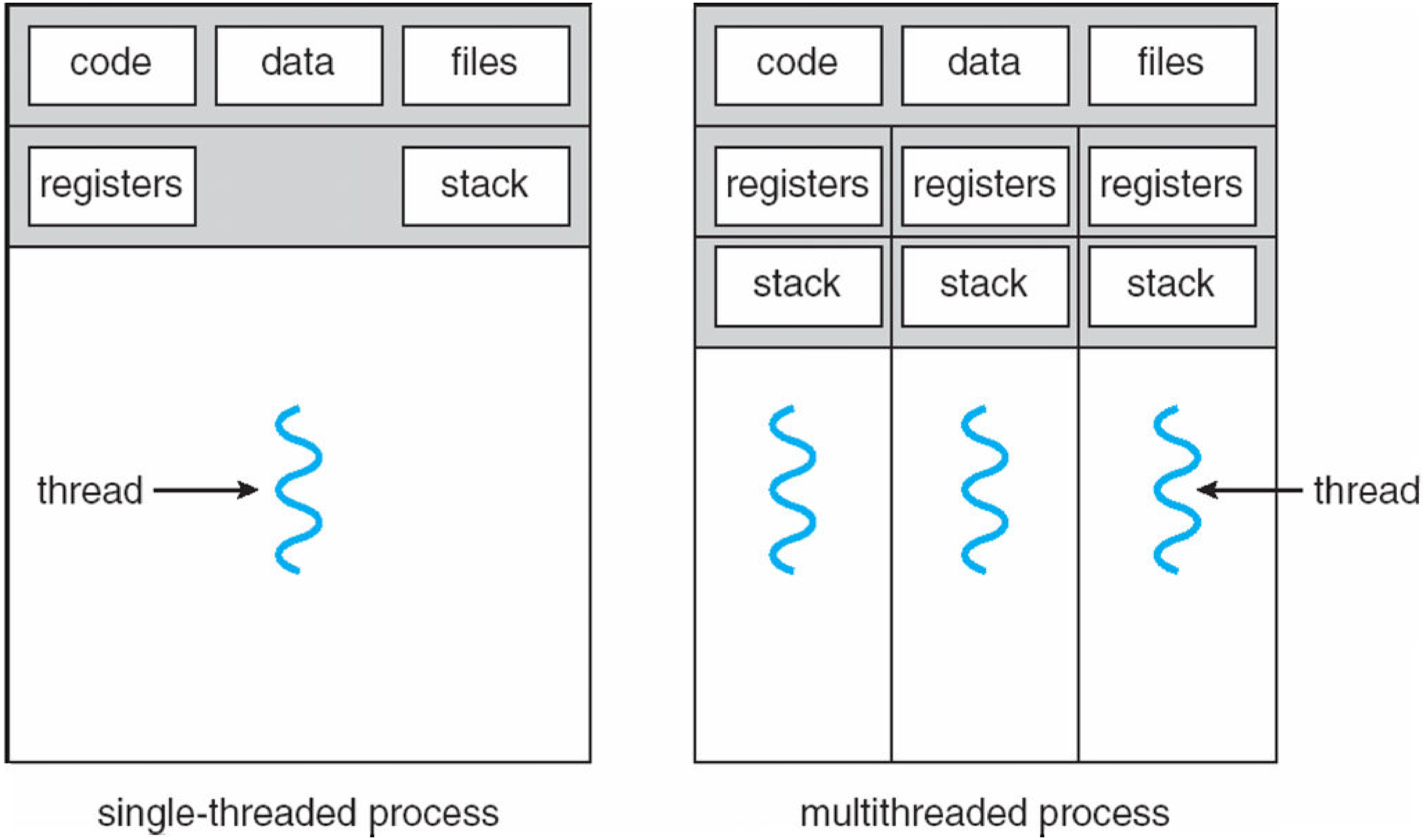
MS Word Process

- As an example, consider the Microsoft Word process.
  - One thread could be used for responding to the user keystrokes.
  - Another thread could could be to display graphics.
  - Yet another thread could perform a grammar and spell check in run-time.
- Important point: all the threads run independently.
- **Advantage of this?**
- **What if we did not have multithreading in this case?**

Web Browser.

- Another example: web browser.
- Scroll a page, while an image is downloading.
- Play an animation with its sound together.
- Print a page, while loading a new page.
- Many other example.
- Many tasks executing at the same time.
- Parallel programmers always in great demand.
- Applications in AI, image processing, video processing.

Single and Multithreaded Processes



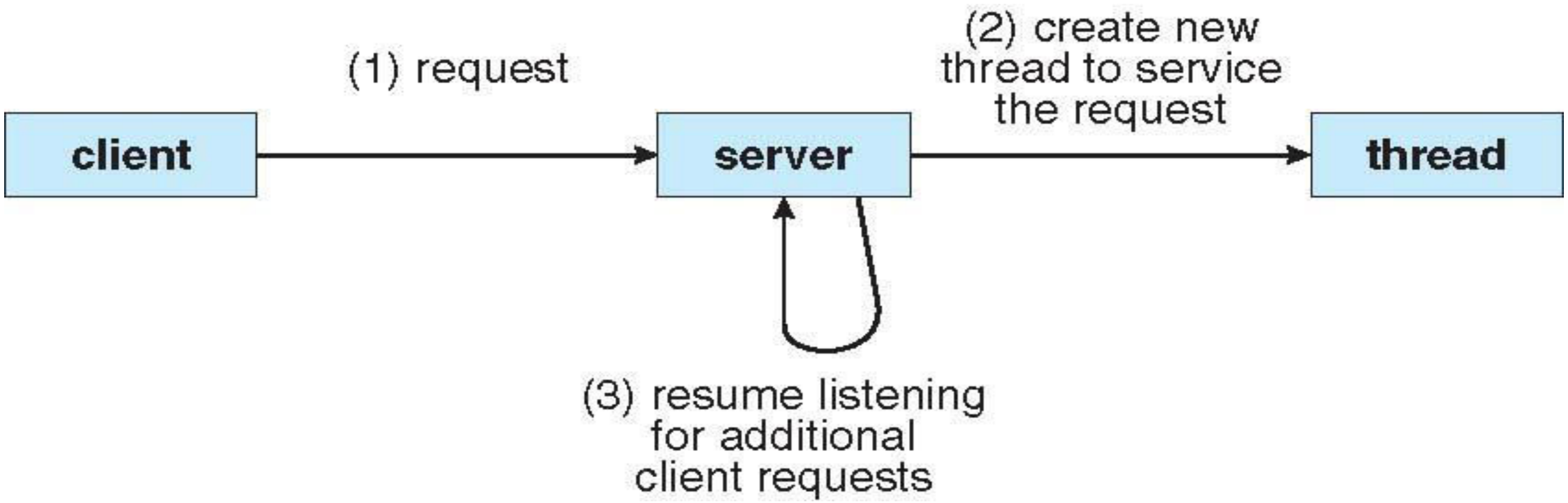
Threads

- Note that in the second case, there are multiple threads of instructions, belonging to the same process.
- Each thread has it’s own:
  - Thread ID.
  - Set of registers (PC and other registers).
  - Stack: for function calls, local variables.
- All threads share the following WRT the process to which they belong to:
  - Code section.
  - Data section.
  - Permissions.
  - Files.

Thread vs. Process

- Another example could be a web server.
  - Separate threads for each client request.
- Single threaded web server.
  - Create a separate process to service each request.
  - Note that all such processes are very similar to each other.
- Multi threaded web server.
  - Create a separate thread to service each request.
- **Creating threads is cheaper than creating processes.**
  - **Lesser work to be done by the OS.**
- All thread share same address space with creator.
  - Separate address space for each process, which is more expensive.
- Certain things like code, data are shared among threads, so cheaper.

Multithreaded Server Architecture



Creating threads vs. processes

- Process includes many things:
  - Address space.
  - Open files.
  - Execution context (PC, SP, registers etc.).
- Creating a new process is costly due to all data structures that must be allocated and initialized.
- Communicating among processes is costly, as most communication goes via the OS.
- Thread communication cheaper, as it is enabled by default.
- Thread creation cheaper, as less data structures to be created.
- Makes sense intuitively, also.

Multi-threaded Kernel

- All modern kernels are multi-threaded.
- If user processes can be multithreaded, why not kernel processes?
- To see all kernel processes on a Linux system: ps –ef.
- Several threads operate in the kernel, performing tasks, such as:
  - Managing devices.
  - Managing memory.
  - Interrupt handling.
- Set of threads for each kernel task.

Why Multithreading?

- **Responsiveness.**
  - Consider the web server application.
  - If the app was single threaded, and one user made a request, the other users would need to wait.
  - If the server is multithreaded (and it is), separate thread for each user, so no waiting (lesser response times).
- **Scalability.**
  - These threads can run in parallel on separate CPUs.
  - This allows designers to scale the application to large number of users (offering decent response times), as the parallelism of the CPUs can be exploited.
- **Economy.**
  - Creating a thread within a process is cheaper than making a separate process out of it.
    - Allocating resources & memory for processes is expensive.
    - Threads share resources & memory of it's process.

Why Multithreading?

- **Resource sharing.**
  - Processes can share resources using: shared memory & message passing.
  - Such techniques must be explicitly arranged by the programmer.
  - In comparison, threads share resources by default, so, no extra work to do by the OS.

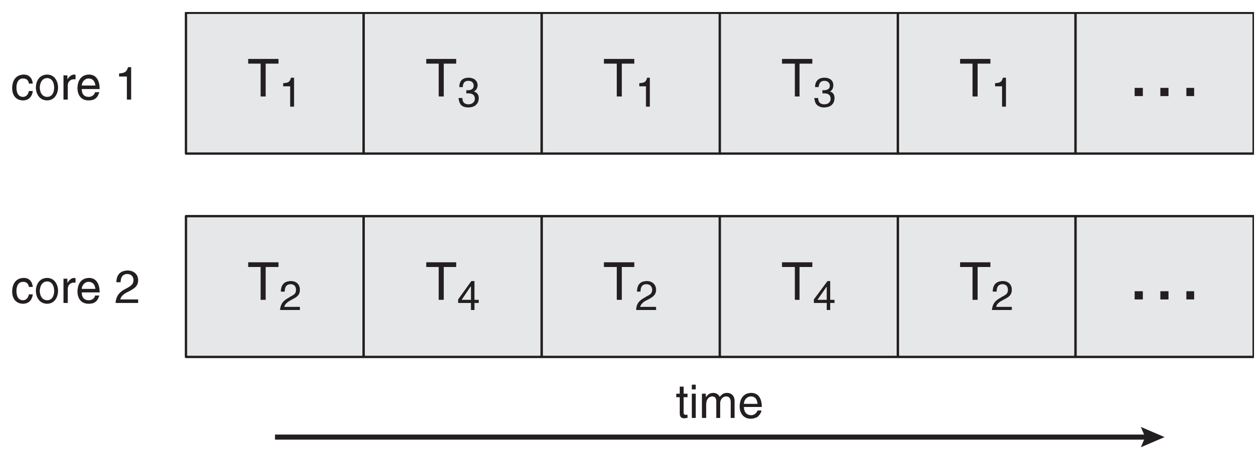
Multicore Programming

- **Multicore**/**multiprocessor** systems are putting pressure on programmers, challenges include:
  - **Dividing activities: identify code that runs in parallel.**
  - **Balance processor load.**
  - **Data splitting among processors.**
  - **Data dependency.**
  - **Testing and debugging (many execution paths possible now).**
- ***Parallelism vs. concurrency.***
- ***Parallelism*** implies a system can run more than one process/thread simultaneously.
- ***Concurrency*** allows many processes/threads to make progress.
  - Single processor / core, scheduler can providing concurrency

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

- **Parallelism on a multi-core system:**



Multicore Programming (Cont.)

- Types of parallelism:
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each.
    - P_1: [0] – [N/2-1], P_2: [N/2+1] – [N-1]. {Array sort}
  - Each CPU works on part of an array, for array addition.
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation.
- As # of threads grows (from tens to thousands), so does architectural support for threading
  - Consider Oracle SPARC T4 server with 8 cores, and 8 hardware threads per core.
  - Multiple threads can be loaded for fast switching.

Pthreads

- Application Programming Interface (API) is available, using which programmers can conveniently write multi-threaded applications.
- API available for both Linux and Windows systems.
- The Linux based API is called Pthreads, where P stands for Posix (Portable Operating Systems Interface).
  - Works on “non-Windows” systems.
- Can be used on Linux, Mac and Solaris devices.
- System calls for various thread operations.
- Let’s look at an example of a C program in which a process creates a separate thread that performs some computation.
  - Calculating the sum of non-negative integers using the following well known formula:
  - $Sum = \sum_{i=1}^{n} i.$

Some Thread Function Calls

- **pthread_create (&tid, &attr, start-function, arg to call start-function with)**
  - **This call is used to create a thread.**
  - Here, tid is the id of the thread.
  - Attr is a pointer to a structure variable that contains the attributes of the thread, such as scheduling priority, state, memory size etc.
  - Upon it's creation, the thread executes start-function.
  - The start-function may be called with an argument (arg).
  - If successful, pthread_create returns a value of 0.
- **pthread_attr_init (&attr)**
  - **Initializes attr with all the default values.**
  - Attributes can be set using appropriate functions.
  - See man pages for description to these functions.
  - Example: pthread_attr_setschedpolicy ( ).

Some Thread Function Calls

- **pthread_join (&tid, ptr)**
  - **Suspends the execution of the calling process until the target thread finishes.**
  - tid is the id of the target thread.
  - If interested in the exit status of target thread, can catch it using ptr.
  - ptr is the address of the memory area that receives the exit status of target thread.
  - If not, second argument is NULL.
- **pthread_exit (void *value_ptr)**
  - **This terminates the called thread.**
  - Resources used by thread can be returned back to the calling process.
  - Function makes available value_ptr to any successful join with the terminating thread.
  - This return value can be used by calling function, if needed.
  - Note that the data type for the pointer is void. This means that it can potentially point to any data type later by casting.

Pthreads Example

```c
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
       sum += i;

    pthread_exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

More Programs

- Let us look at some more C programs that uses multi-threading.
- Program 1:
  - Here, a process creates two threads.
  - Each thread does some work.
- Program 2:
  - Same as program 1, extended to > 2 threads.

End of Chapter 4