Operating System Lab: CS341

LAB 5: Multithreading

Course Instructor: Dr Satendra Kumar

Introduction

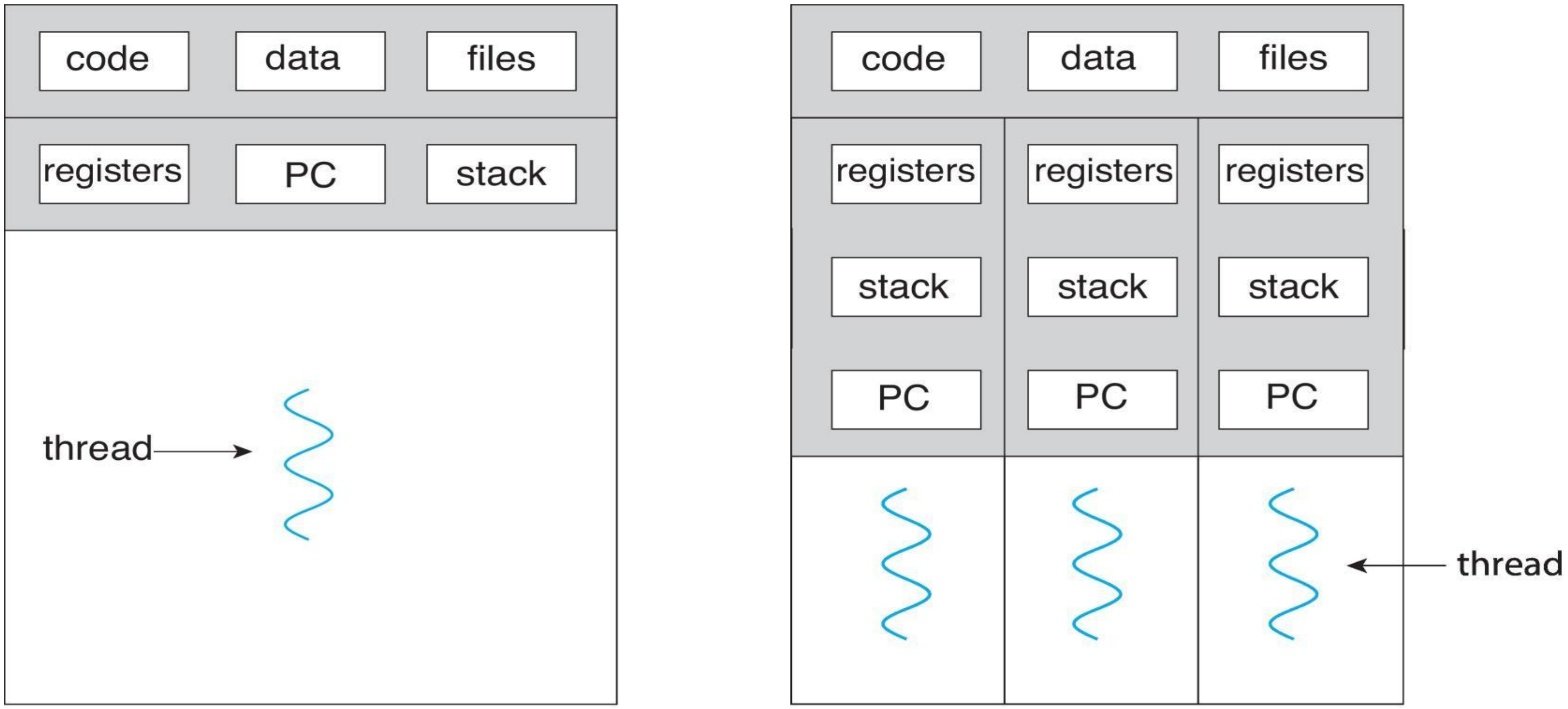
- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

A thread is the smallest unit of process.

A process have memory allocated to it. Within a process multiple threads can access this memory allocated with the process

- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

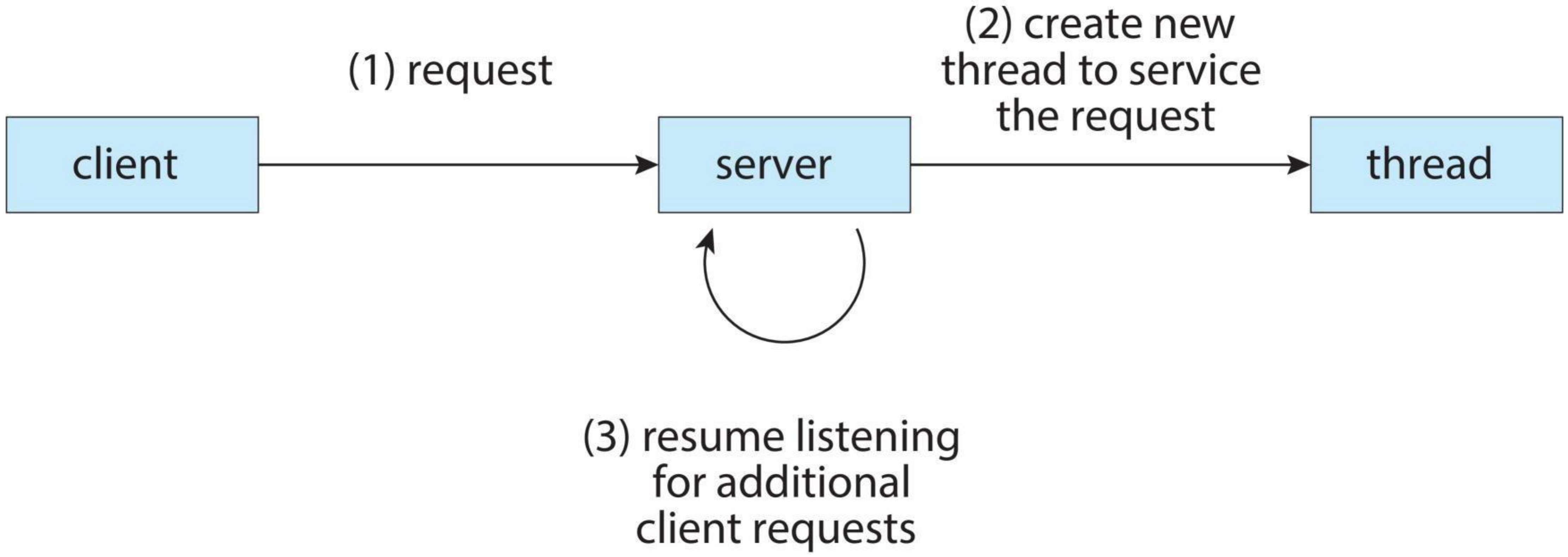
Single Vs Multithreaded processes



single-threaded process

multithreaded process

Multithreaded Server Architecture



Benefits

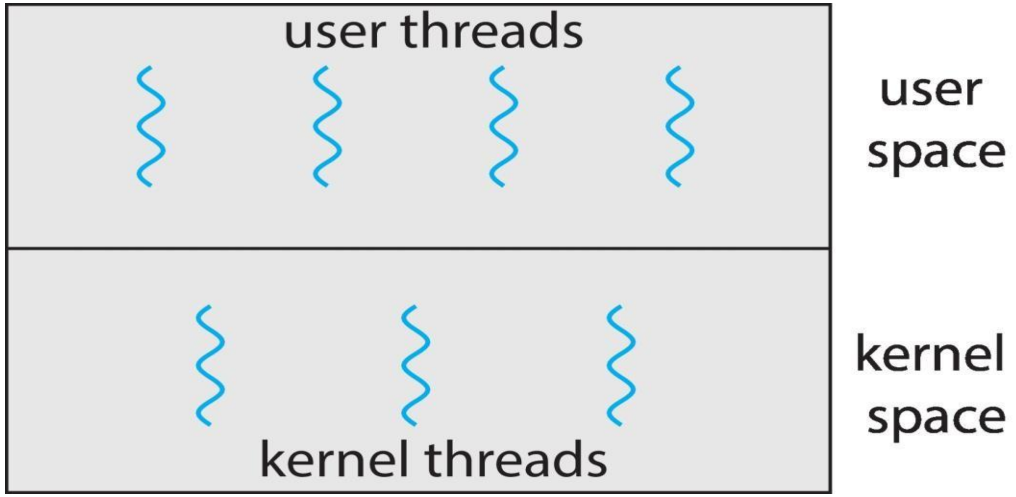
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multicore architectures

Thread Libraries

- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user or in kernel space.
  - User space; API functions implemented solely within user space, with no kernel support.
  - Kernal space; involves system calls, and requires a kernel with
  - thread library support.
  - A few well established primary thread libraries
    - POSIX Pthreads - may be provided as either a user or kernel library
    - Win32 threads - provided as a kernel-level library on Windows systems.
    - Java threads – May be Pthreads or Win32 depending on the OS and hardware the JVM is running.

User Thread and Kernel Threads

- **User threads** - management done by user-level threads library
- **Kernel threads** - Supported by the Kernel
  - Exists virtually in all general purpose OS:
    - Windows, Linux, Mac OS X, iOS, Android
- Even user threads will ultimately need kernel thread support (Why??)



Posix Threads

- POSIX threads, or Pthreads, is a standardized threading library defined by the POSIX (Portable Operating System Interface) standard, specifically the pthread library.
- Pthreads provides a set of functions to create and manage threads in a program, allowing for concurrent execution of code, which can lead to more efficient and responsive applications.
- The **#include <pthread.h>** directive includes the POSIX threads (Pthreads) library in a C program, providing functionality for multithreading. This header file is crucial for creating, managing, and synchronizing threads.
- Key functions:
    - **pthread_create**: Creates a new thread.
    - **pthread_join**: Waits for a thread to finish.
    - **pthread_exit**: Terminates the calling thread.

- **pthread_create:** Creates a new thread and starts executing the specified function in that thread.

  Syntax: int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);

  Parameters:

  - **thread:** Pointer to a pthread_t variable where the thread ID will be stored.
  - **attr:** Thread attributes (use NULL for default attributes).
  - **start_routine:** Function to execute in the new thread.
  - **arg:** Argument passed to the start_routine function.

  Returns: 0 on success, or an error code on failure.

- **Example:**

```
pthread_t thread;
int result = pthread_create(&thread, NULL, start_function, NULL);
if (result != 0) {
    printf("Error creating thread\n");
}
```

- **pthread_join:** Waits for the specified thread to terminate.

  Syntax: int pthread_join(pthread_t thread, void **retval);

  Parameters:

  - **thread:** Thread ID of the thread to wait for.
  - **retval:** Pointer to a location to store the exit status of the thread (use NULL if not needed).

  Returns: 0 on success, or an error code on failure.

- **Example:**

  pthread_join(thread, NULL);

- **pthread_exit:** Exits the calling thread

  Syntax: void pthread_exit(void *retval);

  Parameters:

  - **retval:** Return value of the thread (use NULL if not needed).

  Returns: 0 on success, or an error code on failure.

  Example: pthread_exit(NULL);

Mutexes

Mutexes are used to protect the resources from being accessed by more than one process concurrently

- Mutexes are used to protect shared resources from concurrent access, ensuring that only one thread accesses a critical section at a time.

- Key functions:
  - **pthread_mutex_init**: Initializes a mutex.
  - **pthread_mutex_lock**: Locks a mutex.
  - **pthread_mutex_unlock**: Unlocks a mutex.
  - **pthread_mutex_destroy**: Destroys a mutex.

- **pthread_mutex_init:** Initializes a mutex.

  Syntax: int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)

  Parameters:

  - **mutex:** Pointer to the mutex variable.
  - **attr:** Mutex attributes (use NULL for default attributes).

  Returns: 0 on success, or an error code on failure.

- Mutexes are used to protect shared resources from concurrent access, ensuring that only one thread accesses a critical section at a time.

- Key functions:
  - **pthread_mutex_init**: Initializes a mutex.
  - **pthread_mutex_lock**: Locks a mutex.
  - **pthread_mutex_unlock**: Unlocks a mutex.
  - **pthread_mutex_destroy**: Destroys a mutex.

- **pthread_mutex_init:** Initializes a mutex.

  Syntax: int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)

  Parameters:

  - **mutex:** Pointer to the mutex variable.
  - **attr:** Mutex attributes (use NULL for default attributes).

  Returns: 0 on success, or an error code on failure.

- **Example:**

  pthread_mutex_t mutex;

  pthread_mutex_init(&mutex, NULL);

- **pthread_mutex_lock:** Locks a mutex. If the mutex is already locked, the calling thread blocks until the mutex becomes available.

  Syntax: int pthread_mutex_lock(pthread_mutex_t *mutex);

  Parameters:

  - **mutex:** Pointer to the mutex variable.

  Returns: 0 on success, or an error code on failure.

  Example: pthread_mutex_lock(&mutex);

- **pthread_mutex_unlock:** Unlocks a previously locked mutex.

  Syntax: int pthread_mutex_unlock(pthread_mutex_t *mutex);

  Parameters:

  - **mutex:** Pointer to the mutex variable.

  Returns: 0 on success, or an error code on failure.

  Example: pthread_mutex_unlock(&mutex)

- **pthread_mutex_destroy:** Destroys a mutex.

  Syntax: int pthread_mutex_destroy(pthread_mutex_t *mutex)

  Parameters:

  - **mutex:** Pointer to the mutex variable.

  Returns: 0 on success, or an error code on failure.

  Example: pthread_mutex_destroy(&mutex);

Condition Variables

- Condition variables are used to block a thread until a particular condition is met, allowing threads to wait and be signaled.

- Key functions:
  - **pthread_cond_init**: Initializes a condition variable.
  - **pthread_cond_wait**: Waits for a condition variable to be signaled.
  - **pthread_cond_signal**: Signals a condition variable, waking up one waiting thread.
  - **pthread_cond_broadcast**: Signals all waiting threads.
  - **pthread_cond_destroy**: Destroys a condition variable.

- **pthread_cond_init:** Initializes a condition variable.

  Syntax: int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

  Parameters:

  - **cond:** Pointer to the condition variable.
  - **attr:** Condition variable attributes (use NULL for default attributes).

  Returns: 0 on success, or an error code on failure.

  - **Example:** pthread_cond_t cond;

    pthread_cond_init(&cond, NULL);

- **pthread_cond_wait:** Blocks the calling thread until the specified condition variable is signaled. This function must be called with the mutex locked.

  Syntax: int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

  Parameters:

  - **cond:** Pointer to the condition variable.
  - **mutex:** Pointer to the associated mutex (must be locked before calling).

  Returns: 0 on success, or an error code on failure.

  - **Example:** pthread_cond_wait(&cond, &mutex);

- **pthread_cond_signal:** Unblocks one thread waiting on the specified condition variable.

  Syntax: int pthread_cond_signal(pthread_cond_t *cond);

  Parameters:

  - **cond:** Pointer to the condition variable.

  Returns: 0 on success, or an error code on failure.

  - **Example:** pthread_cond_signal(&cond);

- **pthread_cond_broadcast:** Unblocks all threads waiting on the specified condition variable.

  Syntax: int pthread_cond_broadcast(pthread_cond_t *cond);

  Parameters:

  - **cond:** Pointer to the condition variable.

  Returns: 0 on success, or an error code on failure.

  - **Example:** pthread_cond_broadcast(&cond);

- **pthread_cond_destroy:** Destroys a condition variable.

  Syntax: int pthread_cond_destroy(pthread_cond_t *cond);

  Parameters:

  - **cond:** Pointer to the condition variable.

  Returns: 0 on success, or an error code on failure.

  - **Example:** pthread_cond_destroy(&cond);

- **SortParam Struct:** A structure that holds parameters for the merge_sort function, including the array and the indices of the subarray to sort.

  Syntax: typedef struct {

  int *array;

  int left;

  int right;} SortParams;

  Example: SortParams params = {array, 0, n - 1};

- **malloc:** Allocates memory dynamically on the heap.

  Syntax: void *malloc(size_t size);

  Parameters:

  - **size_t size:** Number of bytes allocated.

  - **Example:** int *array = (int *)malloc(n * sizeof(int));

- **free:** Frees dynamically allocated memory, preventing memory leaks.

  Syntax: void free(void *ptr);

  Parameters:

  - **void *ptr:** Pointer to the memory block to be freed.

  - **Example:** free(array);

- **Error Handling with perror and fprintf:** Error handling is used to report issues, such as memory allocation failures or invalid input, and gracefully exit the program.

  Syntax: void perror(const char *s);

  int fprintf(FILE *stream, const char *format, ...);

  Example: if (array == NULL) {

  perror("Failed to allocate memory");

  return EXIT_FAILURE;}