



SLIIT

Discover Your Future

IT2060/IE2061

Operating Systems and System Administration

Lecture 04

Introduction to Threads

U. U. Samantha Rajapaksha

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

Samantha.r@slit.lk



SLIIT
FACULTY OF COMPUTING

Motivation

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

Execution of one main function is called single set of threads

Threads

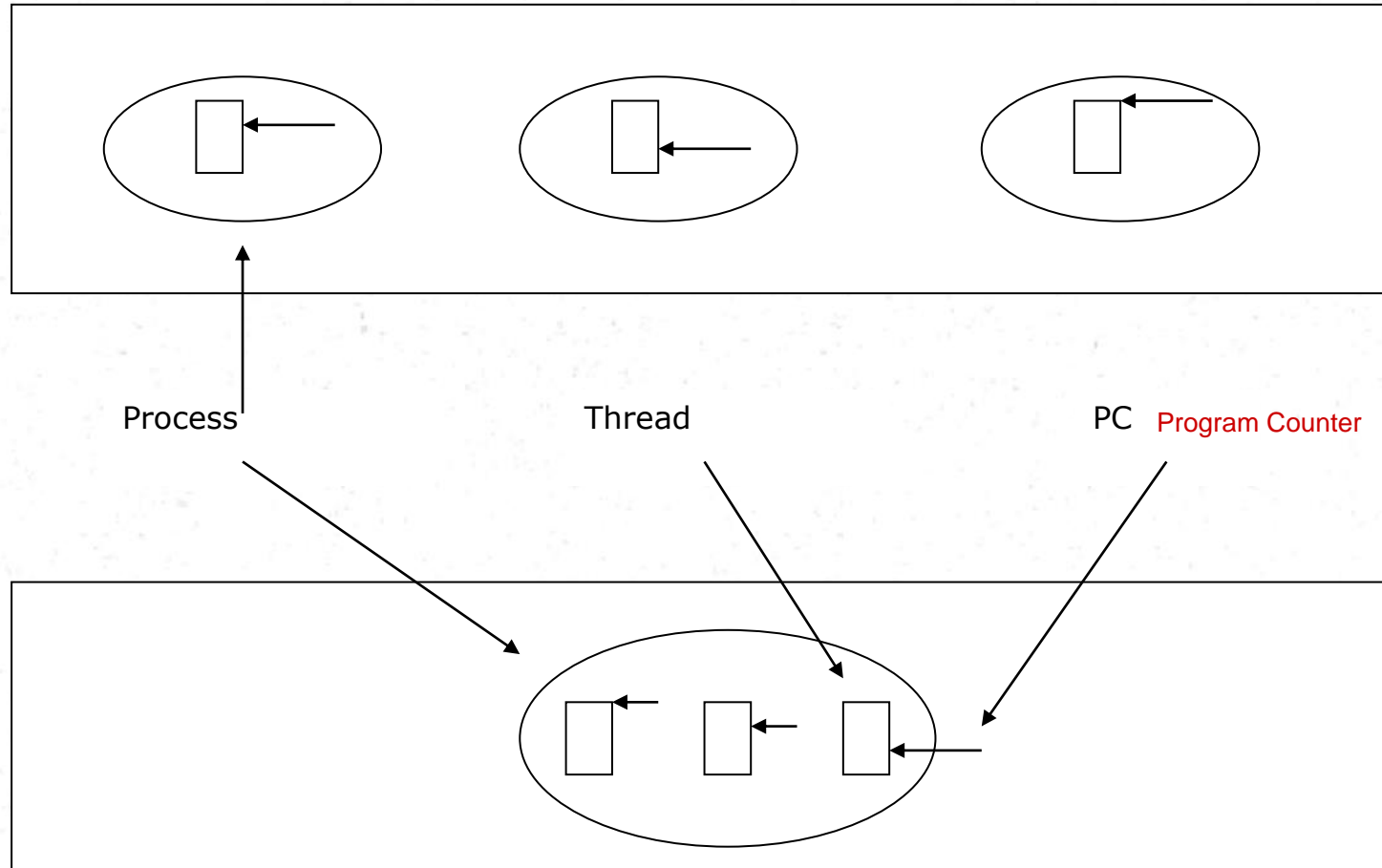
- A thread of control is an independent sequence of execution of program code in a process.
- A thread (or lightweight process) is a basic unit of CPU utilization.
 - A traditional process (or heavyweight process) is equal to a task with one thread.
- A traditional process has a single thread that has sole possession of the process's memory and other resources → context switch becomes performance bottleneck
 - Threads are used to avoid the bottleneck.
 - Threads share all the process's memory, and other resources.
- Threads within a process:
 - are generally invisible from outside the process.
 - are scheduled and executed independently in the same way as different single-threaded processes.
- On a multiprocessor, different threads may execute on different processors
 - On a uni-processor, threads may interleave their execution arbitrarily.

Threads (cont.)

- Threads operate, in many respects, in the same manner as processes:
 - Threads can be in one of several states: ready, blocked, running, or terminated, etc.
 - Threads share CPU; only one thread at a time is running.
 - A thread within a process executes sequentially, and each thread has its own PC and Stack.
 - Thread can create child threads, can block waiting for system calls to complete
 - if one thread is blocked, another can run.
- One major different with process: threads are not independent of one another
 - all threads can access every address in the task → a thread can read or write any other thread's stack. process
 - There is no protection between threads (within a process);
 - however this should not be necessary since processes may originate from different users and may hostile to one another while threads (within a process) should be designed (by same programmer) to assist one another.

Threads (cont.)

Three processes with one thread each

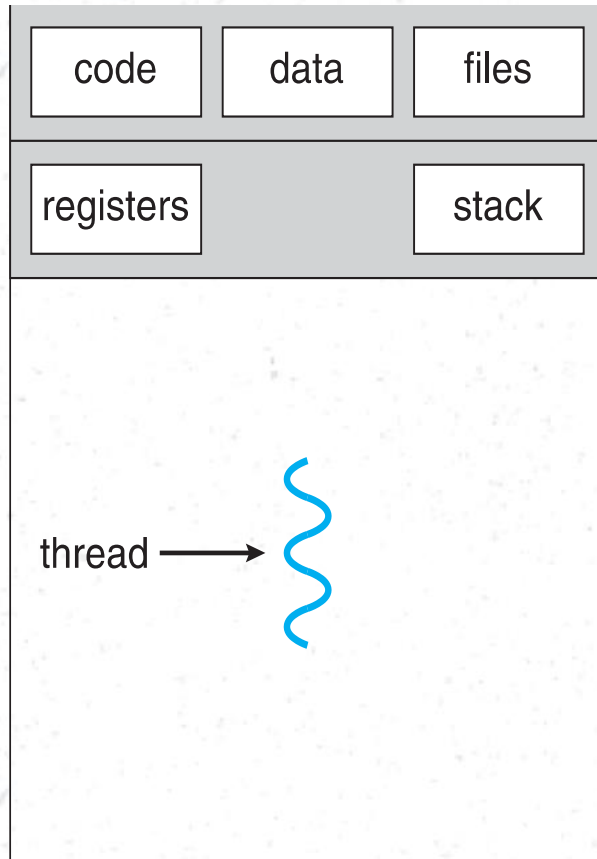


One process with three threads

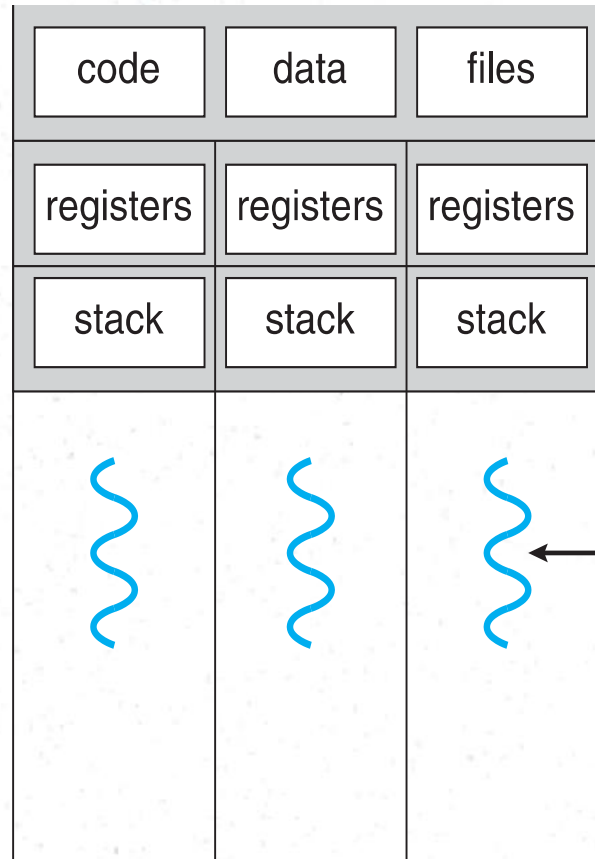
Benefits

- **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 multiprocessor architectures

Single and Multithreaded Processes



single-threaded process



multithreaded process

User Threads and Kernel Threads

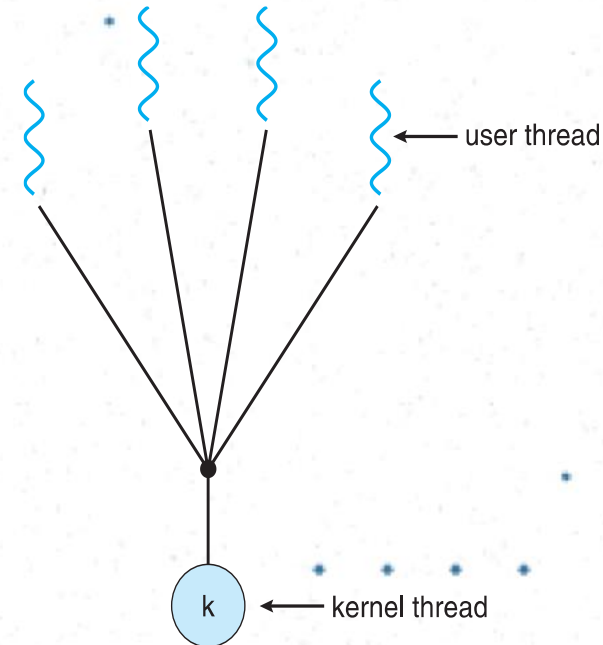
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel threads** - Supported by the Kernel They are in the OS
- Examples – virtually all general purpose operating systems, including:
 - Windows
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

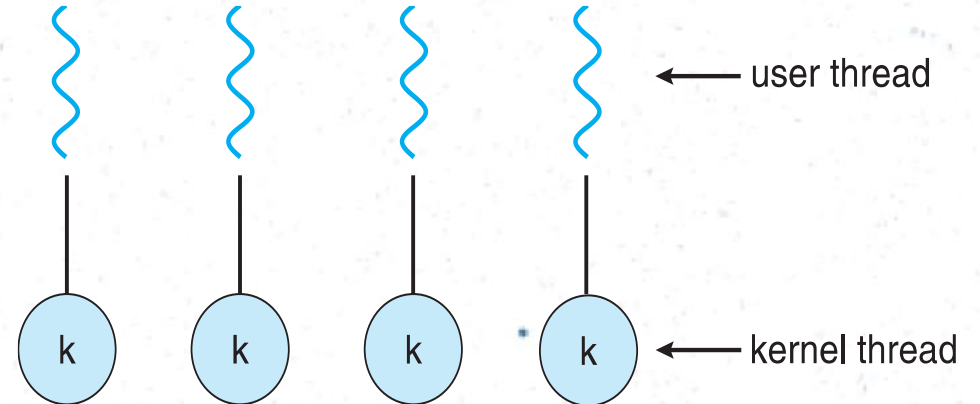
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - **Solaris Green Threads**
 - **GNU Portable Threads**



One-to-One

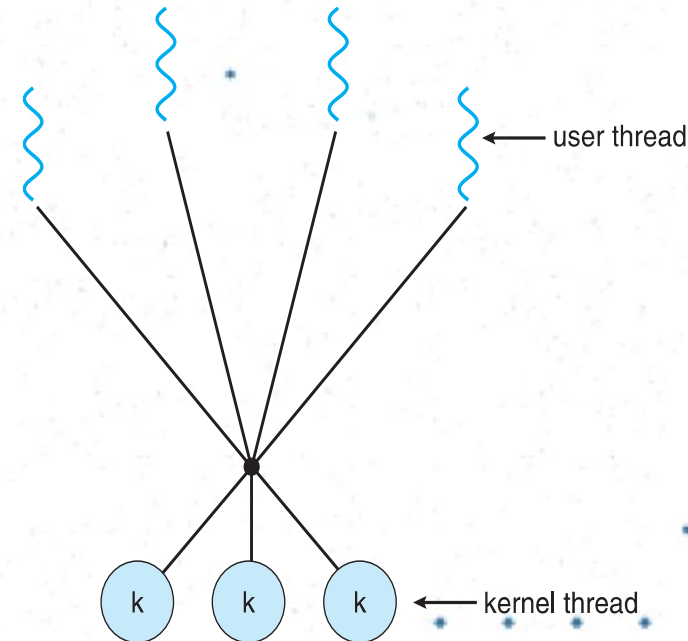
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux
 - Solaris 9 and later

if you increase user threads, kernel threads also increase
which is a resource leak



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification*, not *implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool
 - Separating task to be performed from mechanics of creating task allows different strategies for running task
 - i.e.Tasks could be scheduled to run periodically

Signal Handling

A signal is generated by an event, and it should be sent to the process, and it should be handled by the process

- n **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- n A **signal handler** is used to process signals
 1. Signal is generated by particular event
 2. Signal is delivered to a process
 3. Signal is handled by one of two signal handlers:
 1. default
 2. user-defined
- n Every signal has **default handler** that kernel runs when handling signal
 - | **User-defined signal handler** can override default
 - | For single-threaded, signal delivered to process

Signal Handling (Cont.)

All these 4 cases are applicable

n Where should a signal be delivered for multi-threaded?

- | Deliver the signal to the thread to which the signal applies
- | Deliver the signal to every thread in the process
- | Deliver the signal to certain threads in the process
- | Assign a specific thread to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Thread Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) { /* thread main */
    printf("Hello Thread\n");
    return 0;
}

int main (void) {
    int i;
    pthread_t tid[NUM_THREADS];
    for (i = 0; i < NUM_THREADS; i++) { /* create/fork threads */
        pthread_create(&tid[i], NULL, hello, NULL);
    }
    for (i = 0; i < NUM_THREADS; i++) { /* wait/join threads */
        pthread_join(tid[i], NULL);
    }
    return 0;
}
```

```

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

/* Producer/consumer program illustrating conditional variables */ /* Size of shared buffer */ #define BUF_SIZE 3

int buffer[BUF_SIZE]; /* shared buffer */

int add=0; /* place to add next element */

int rem=0; /* place to remove next element */

int num=0; /* number elements in buffer */

pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER; /* mutex lock for buffer */

pthread_cond_t c_cons=PTHREAD_COND_INITIALIZER; /* consumer waits on this cond var */

pthread_cond_t c_prod=PTHREAD_COND_INITIALIZER; /* producer waits on this cond var */

void *producer(void *param);

void *consumer(void *param);

main (int argc, char *argv[])

{
    pthread_t tid1, tid2; /* thread identifiers */

    int i; /* create the threads; may be any number, in general */

    if (pthread_create(&tid1,NULL,producer,NULL) != 0) {
        fprintf (stderr, "Unable to create producer thread\n"); exit (1); }

    if (pthread_create(&tid2,NULL,consumer,NULL) != 0) {
        fprintf (stderr, "Unable to create consumer thread\n"); exit (1);

    }

    /* wait for created thread to exit */

    pthread_join(tid1,NULL);

    pthread_join(tid2,NULL);

    printf ("Parent quitting\n"); }

}

```

Buffer is created for easier communication between producer and consumer

and when the buffer is full, producer has to wait
when the buffer is zero, consumer has to wait

when the producer has finished producing after unlocking
the buffer through mutex. it sends a message to consumer.


```

/* Produce value(s) */
void *producer(void *param)
{
    int i;
    for (i=1; i<=20; i++) {
        /* Insert into buffer */
        pthread_mutex_lock (&m);
        if (num > BUF_SIZE) exit(1);          /* overflow */
        while (num == BUF_SIZE)               /* block if buffer is full */
            pthread_cond_wait (&c_prod, &m);
        /* if executing here, buffer not full so add element */
        buffer[add] = i;
        add = (add+1) % BUF_SIZE;
        num++;
        pthread_mutex_unlock (&m);
        pthread_cond_signal (&c_cons);
        printf ("producer: inserted %d\n", i); fflush (stdout);
    }
    printf ("producer quitting\n"); fflush (stdout);
}

```

```
/* Consume value(s); Note the consumer never terminates */
```

```
void *consumer(void *param)
```

```
{
```

```
    int i;
```

```
    while (1) {
```

```
        pthread_mutex_lock (&m);
```

```
        if (num < 0) exit(1); /* underflow */
```

```
        while (num == 0)          /* block if buffer empty */
```

```
            pthread_cond_wait (&c_cons, &m);
```

```
        /* if executing here, buffer not empty so remove element */
```

```
        i = buffer[rem];
```

```
        rem = (rem+1) % BUF_SIZE;
```

```
        num--;
```

```
        pthread_mutex_unlock (&m);
```

```
        pthread_cond_signal (&c_prod);
```

```
        printf ("Consume value %d\n", i); fflush(stdout);
```

```
    }
```

```
}
```

Thank you