Chapter 4: Threads

CS303 18sep2018

Motivation

- Threads run within an application
- An “Application” if running typically involves following tasks on the same set of data:
  - Being rendered on the display, style, face, etc.
  - Data access/modification locally
  - Data access/modification across network
  - Additional tasks on the same data

These “Multiple tasks” with the application can be implemented by separate threads

  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

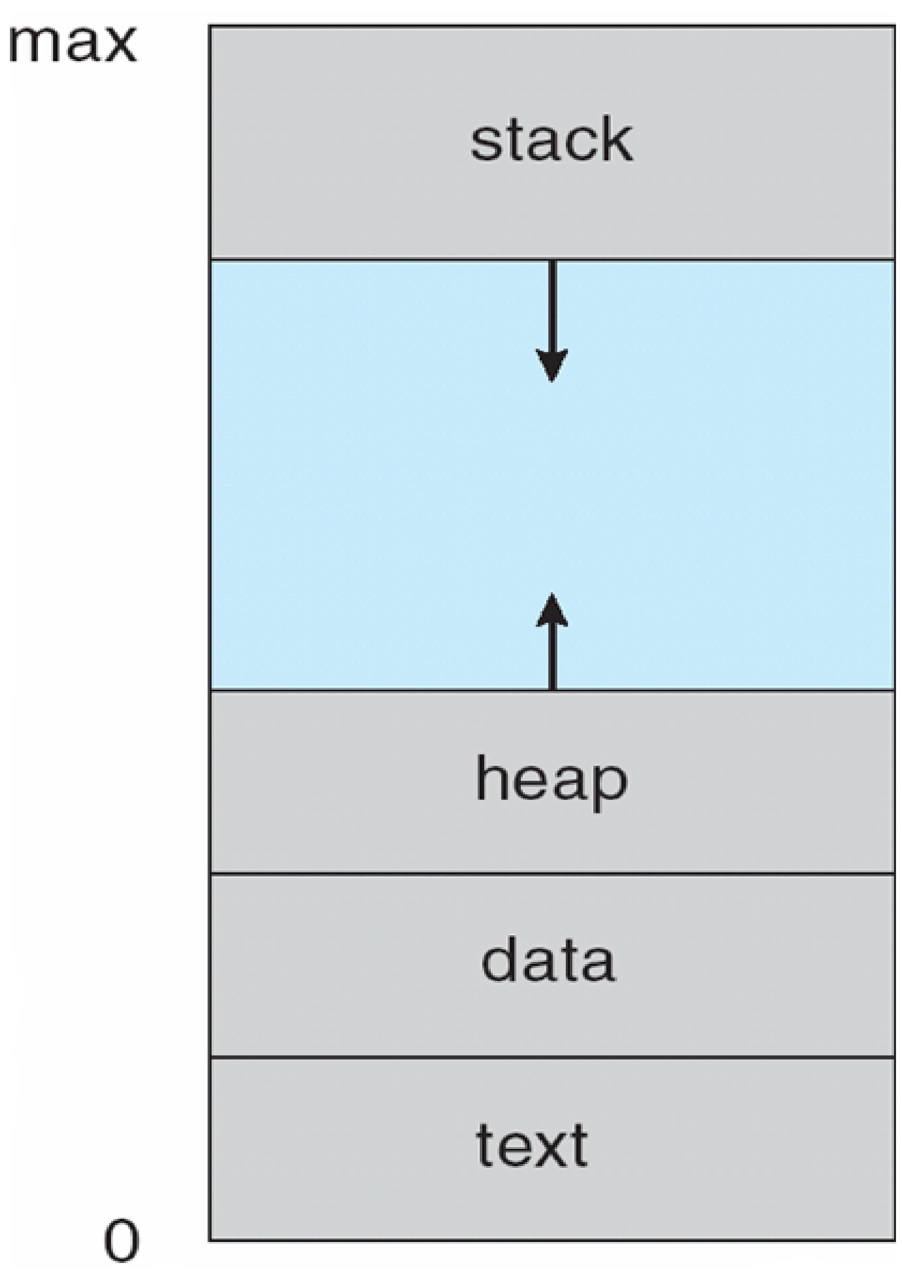
    - We could achieve similar multitasking with the paradigm of interacting processes
      - All sharing the same data set

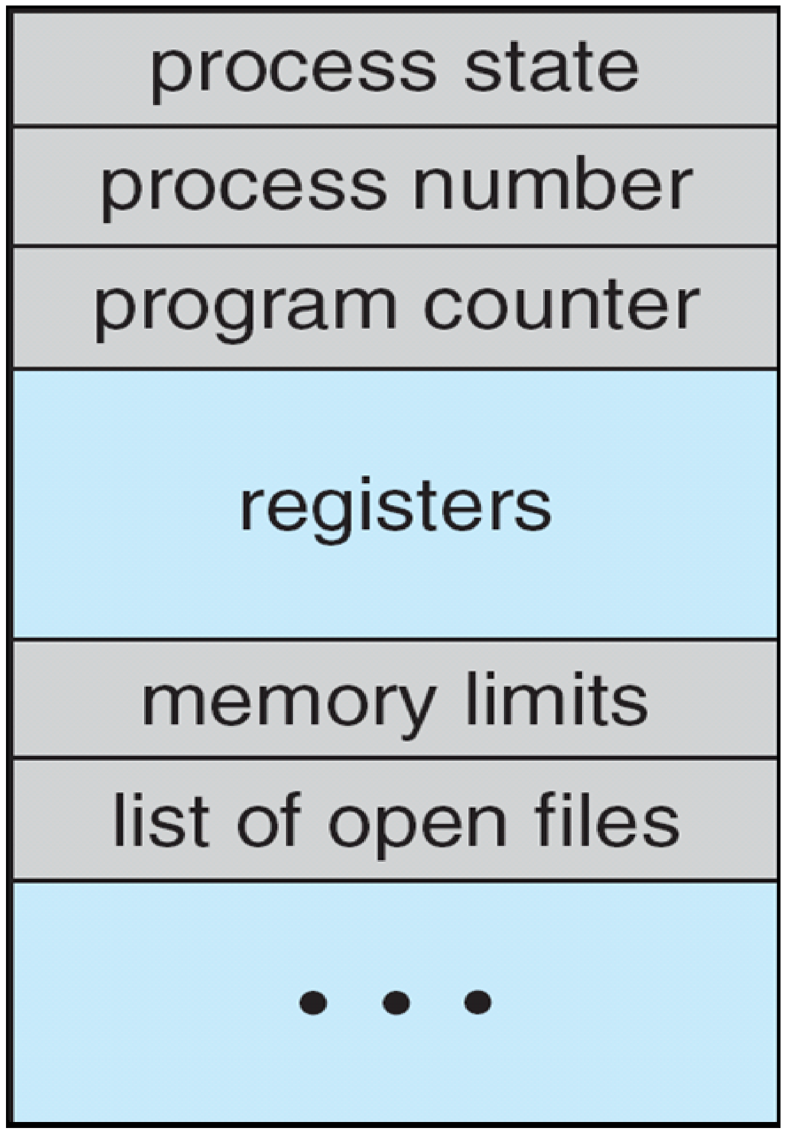
- But: Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Motivation (2)

- We could gain in terms of performance, if exploit:
  - Data parallelism: to carryout same operation on different sets of data
  - Task parallelism: sto carryout different operations on the same set of data

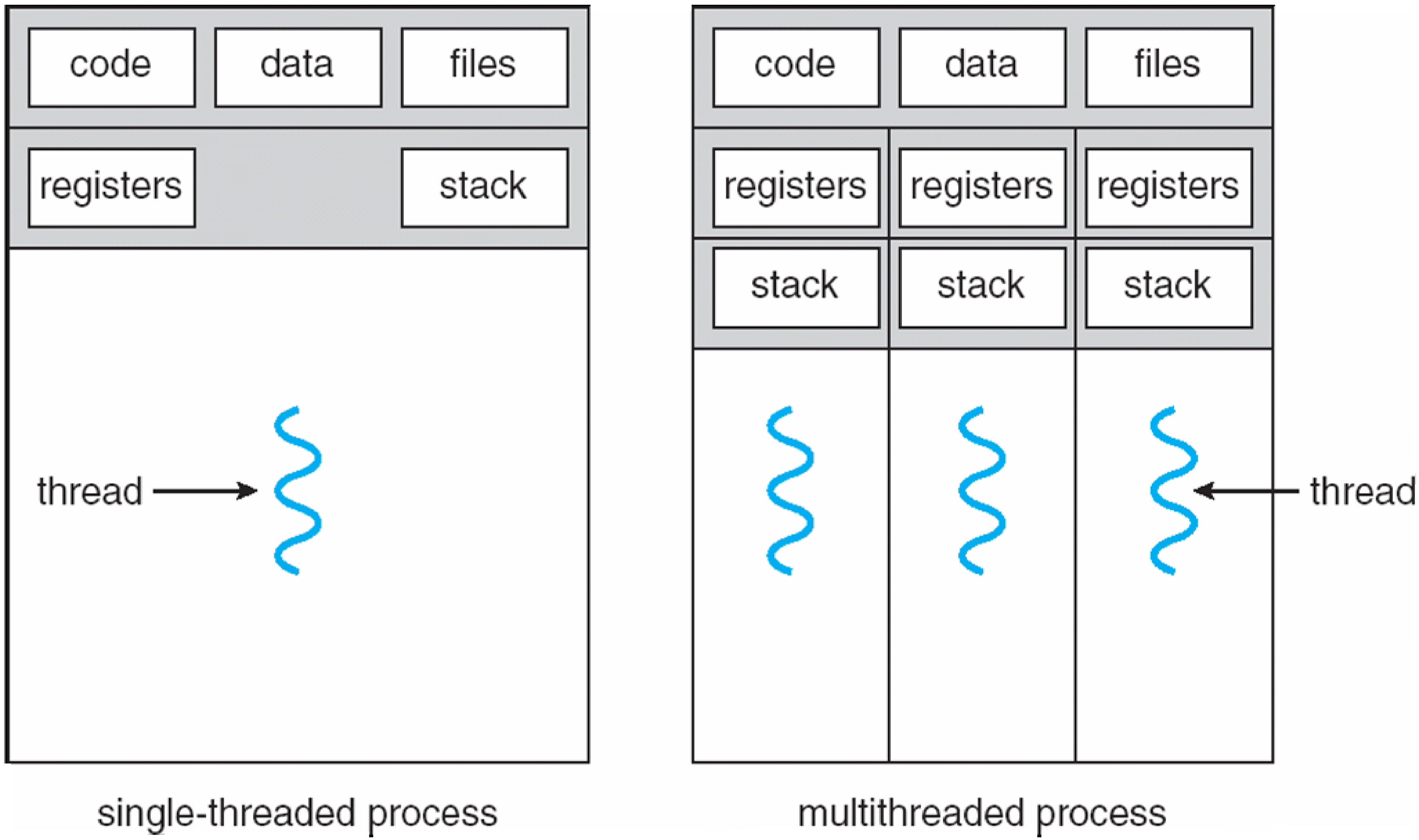
Recall Process Memory Map and PCB



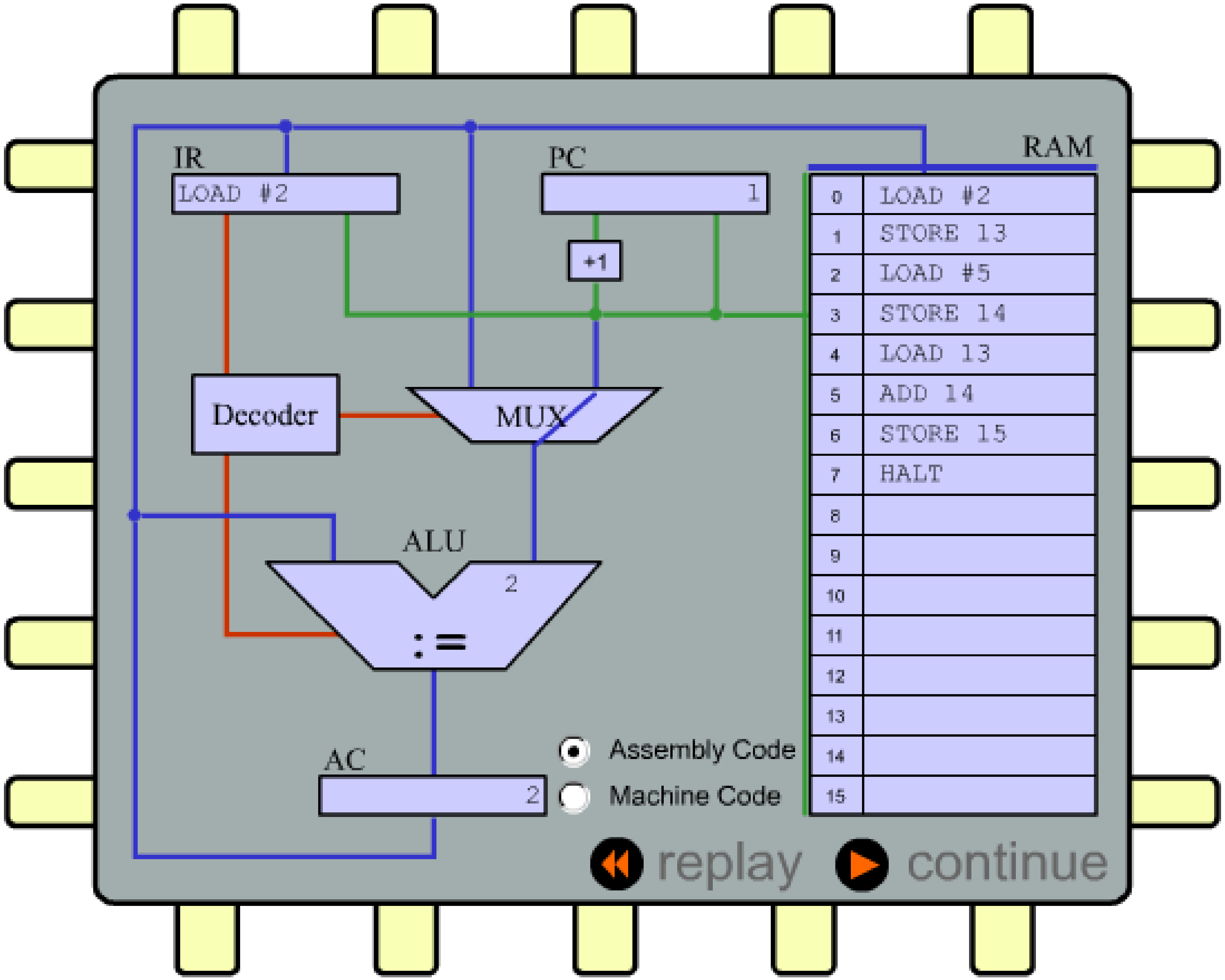


CPU Registers: GPR, Segment registers, SPR, Flag registers

Single and Multithreaded Processes



Recall Process Execution inside the CPU



Thread Control Block

TCB: a data structure in the operating system kernel which contains thread-specific information needed to manage it

- **TCB contents:**
  - **Status: specific to the Thread State Transition Diagram**
    - **Running on CPU**
    - **Ready**
    - **Suspended**
    - **Blocked**
    - **...**
  - **Attributes: ...**
  - **Context: stores the context as contained in the thread registers**
  - **Thread parameters: start functions, parameters stack size, ..**
  - **Stack address: pointer to the stack for this thread**
  - **...**

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
- 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)

Pthreads Example

```
#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.

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Java Multithreaded Program

```
class Sum
{
   private int sum;

   public int getSum() {
    return sum;
   }

   public void setSum(int sum) {
    this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
   }

   public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
       sum += i;
    sumValue.setSum(sum);
   }
}
```

Java Multithreaded Program (Cont.)

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
       else {
        // create the object to be shared
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try {
           thrd.join();
           System.out.println
                   ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

Figure 4.11 Java program for the summation of a non-negative integer.

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- **Thread cancellation** of **target thread**
  - Asynchronous or deferred
- **Signal** handling
  - Synchronous and asynchronous

Threading Issues (Cont.)

- **Thread pools**
- **Thread-specific data**
  - Create Facility needed for data private to thread
- **Scheduler activations**

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?

Thread Cancellation

- Terminating a thread before it has finished

- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately.
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.
    - At cancellation points!

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- 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
- Options:
  - 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 Pools

- Create a number of threads in a pool where they await work
- Typically, there are two paradigms: fixed pool-size vs. dynamic pool-size
  - Stable load
  - Dynamic load
- This number of threads is dependent on the:
    - Hardware
      - Number of CPUs/cores in the system
      - The amount of Physical memory
      - Anticipated load

- 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

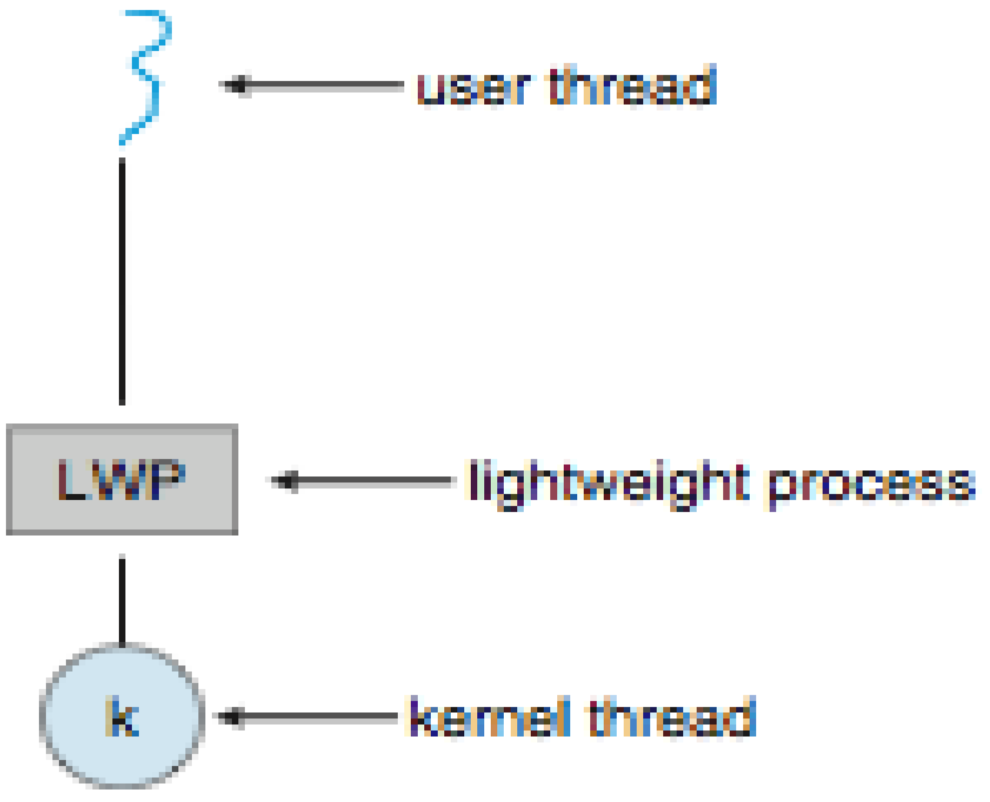
Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- User-threading library schedules user-threads onto available LWPs
- Kernel schedules k-threads on the available hardware threads/cores
- If both schedulers coordinate/communicate optimal resource utilisation is possible
  - Scheduler activation is a coordination technique betweer UTL and Kernel

- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library

- This communication allows an application to maintain the correct number kernel threads
  - How?

Lightweight Processes



Operating System Examples

- Linux Thread

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
- `struct task_struct` points to process data structures (shared or unique)

Linux Threads

- `fork()` and `clone()` system calls
- Doesn't distinguish between process and thread
  - Uses term *task* rather than thread
- `clone()` takes options to determine sharing on process create
- `struct task_struct` points to process data structures (shared or unique)

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.