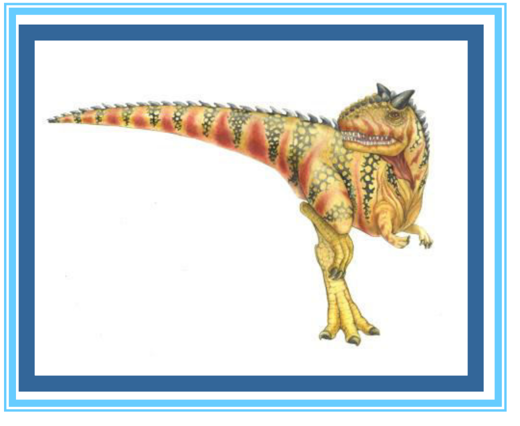
Section 4: Threads



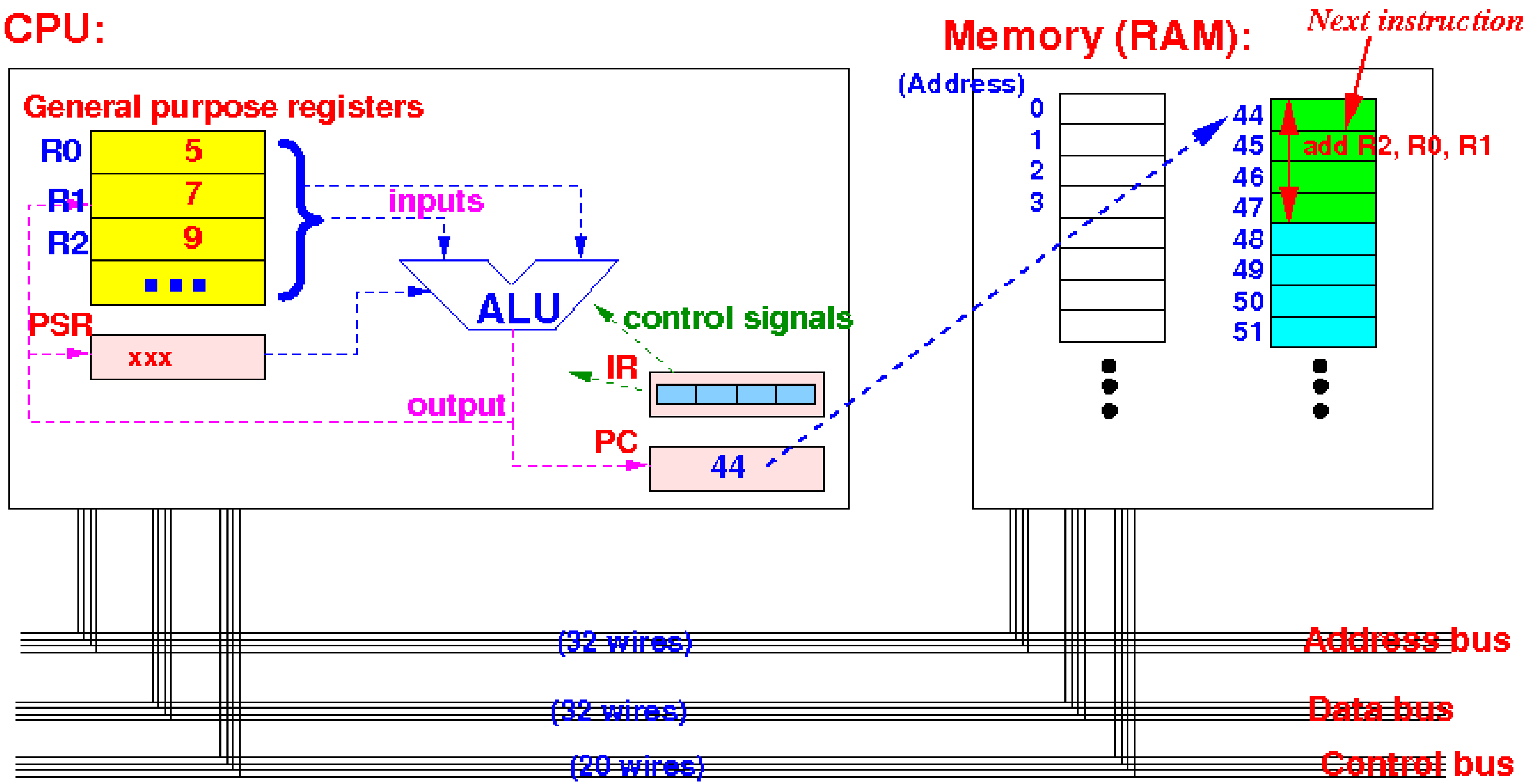
Outline

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Multicore Programming
- Operating System Examples

Objectives

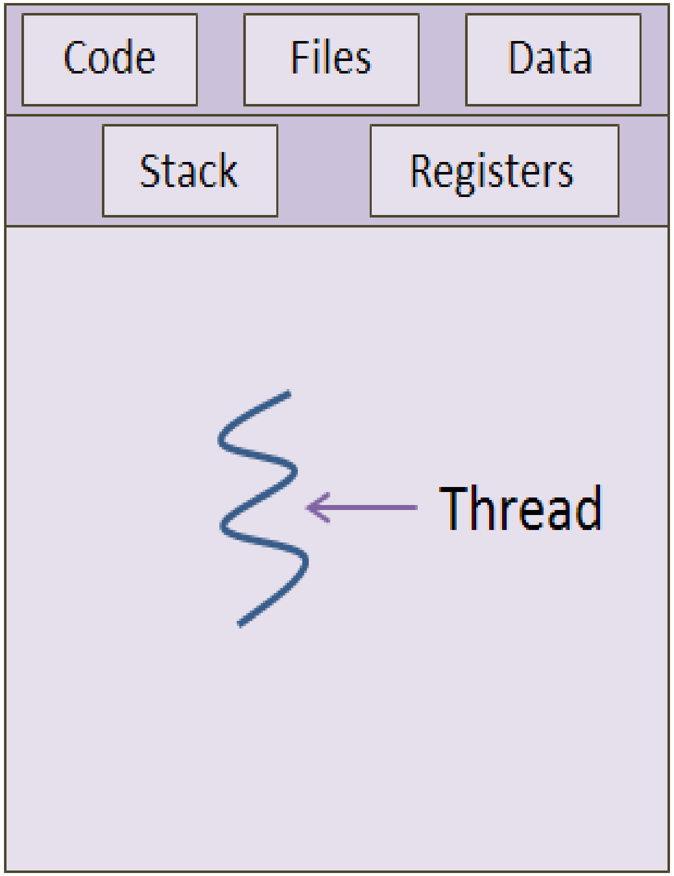
- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Describe how the Windows and Linux operating systems represent threads

Instruction Execution Cycle



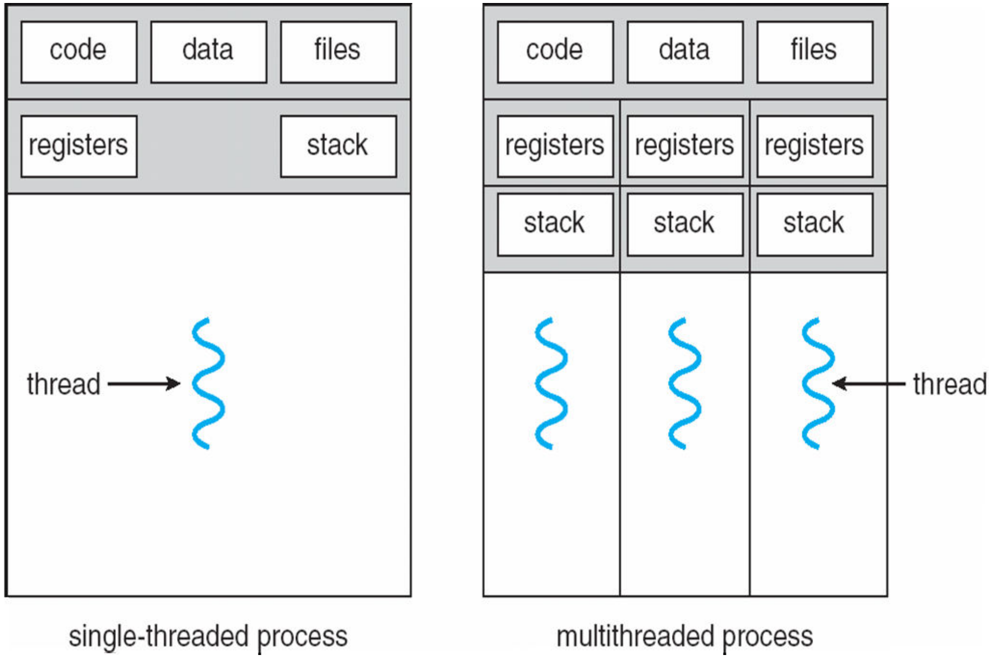
Threads

- When a program executes, the CPU uses the program counter to determine which instruction to execute next.
- The resulting stream of instructions is called the program's *thread of execution*
- The stream of instructions can be represented by the sequence of addresses assigned to the program counter during the execution of the program's code.
- A process may execute statements 245, 246 and 247 in a loop. Its thread of execution can be represented as 245, 246, 247, 245, 246, 247, 245, 246, 247 . . . ,
- Each process executes a sequence of instructions which appears to the process as an uninterrupted stream of addresses



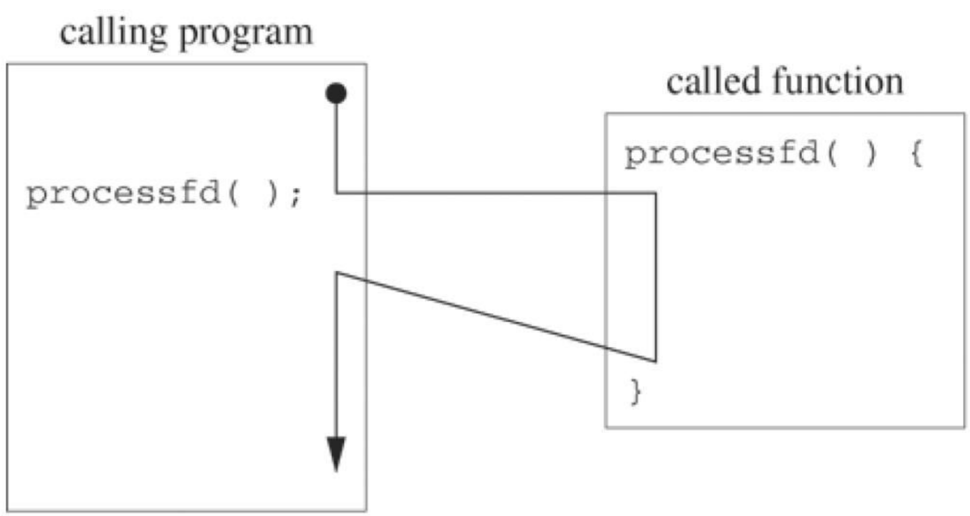
Multi-threaded processes

- Multi-threading means to have multiple threads within the same process.
- This is achieved by dividing the single thread into different sequences of addresses
- Each thread has its own execution stack, program counter value, register set and state.



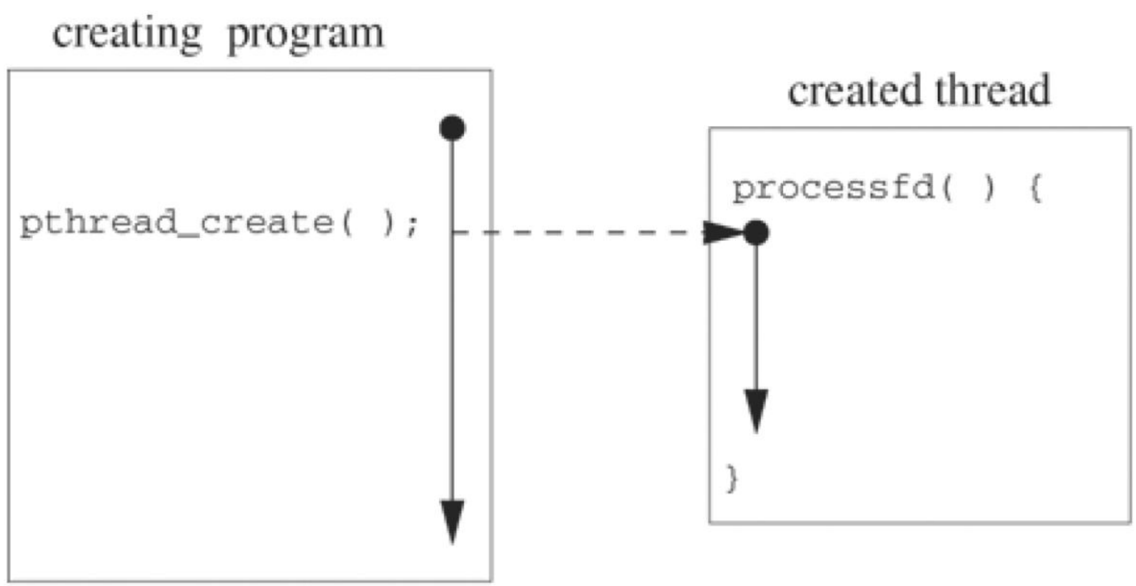
One thread execution

- The figure below illustrates a call to a function (processfd) within the same thread of execution.
- The calling mechanism creates an activation record on the stack
- The thread of execution jumps to processfd when the calling mechanism writes the starting address in the processor's program counter.
- The `return` statement copies the return address that is stored in the activation record into the program counter, causing the thread of execution to jump back to the calling program.



Two threads for the same program

- Next figure illustrates the creation of a separate thread to execute the processfd function. The `pthread_create` call creates a new thread with its own value of the program counter, its own stack and its own scheduling parameters.
- The thread executes an independent stream of instructions, never returning to the point of the call.
- The calling program continues to execute concurrently. In contrast, when `processfd` is called as an ordinary function, the caller's thread of execution moves through the function code and returns to the point of the call, generating a single thread of execution rather than two separate



Threads package

- Instructions for the management of threads are contained in a “thread package” (thread library)
- Thread operations include:
  - thread creation,
  - termination,
  - synchronization (joins,blocking),
  - scheduling,
  - data management
- There are 3 main thread libraries:
  - POSIX threads (are called *pthreads* because all the thread functions start with `pthread`)
  - Win32 threads: Library for Windows
  - Java threads

Some POSIX threads functions

- `pthread_create` create a thread
- `pthread_join` wait for a thread
- `pthread_exit` exit a thread without exiting process
- `pthread_self` find out own thread ID

- Calling `pthread_create`

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

first argument receives the thread id (tid), second are the attributes of the thread (such as scheduling priority), the third is the name of the function to execute by this thread, and last are the parameters of the function

Pthreads Example

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

#include <stdlib.h>

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

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

   /* set the default attributes of the thread */
   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);
}
```

Pthreads Example (Cont.)

```
/* The thread will execute 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);
}
```

Pthreads Example (Cont.)

- The program above takes an input parameter *n* and adds the sequence of numbers from 0 to *n*.
- It has two threads: the initial (or parent) thread in main() and the summation (or child) thread performing the summation operation in the runner() function. (see Linux thread1.c)
- Pthreads programs must include the pthread.h header file.
- The statement pthread_t tid declares the variable that will hold the id of the thread to be created.
- The summation thread "runner" has local variables that can only used by that thread
  - It has also a global variable "sum" which is shared by all the threads: Static (global) variables are shared by all threads
- The summation thread terminates when it calls the function pthread_exit(0).
- Once the summation thread has returned, the parent thread will output the value of the shared static variable *sum*.
- ps –T to see all the threads that are running, SPID are the threads ID
- To compile with threads: gcc thread.c -o thread –lpthread

Pthread example (cont.)

- This program follows the thread create/join strategy, whereby after creating the summation thread, the parent thread wait for it to terminate by calling the pthread join() function.

- Each thread has a set of attributes, including stack size and scheduling information.
  - The declaration pthread attr_t attr defines the storage for the attributes of the thread.
  - Because the attributes are not explicitly set, the default attributes are passed using the function call pthread_attr_init(&attr).

Waiting for a thread: pthread_join

- The pthread_join function causes the caller to wait for the specified thread to exit, similar to waitpid at the process level.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

where first parameter is the thread id and the second one provides a location for a pointer to the return status that the target thread passes to pthread_exit or return. If value_ptr is NULL, the caller does not retrieve the target thread return status.

```
if (error = pthread_join(tid, &exitcodep))
fprintf(stderr, "Failed to join thread: %s\n",
strerror(error));
else
fprintf(stderr, "The exit code was %d\n", *exitcodep);
```

Thread termination after exit

- When a thread exits, by calling pthread_exit(), it does not release its resources. The command *pthread_exit() only* signal the parent thread that the child thread has finished

- It is only when the parent executes pthread_join() that the resources of child thread are released (such as memory for the run time stack)

- However, as shown in the texit1.c, the thread ID is removed from the process table immediately after the thread has exited

- Child threads that have exited but for which the join has not yet been executed by the parent thread have similar status as a zombie process except they are no longer in the process table

- If parent threads never execute join, then we have memory leak, it may come to a point where threads can no longer be created

- Do ps –T to get the thread id in linux

texit1.c

```
int sum;void *runner(void *param);
int main(int argc, char *argv[]) {
   pthread_t tid;
   pthread_create(&tid, &attr, runner, argv[1]);
   sleep(10);  /*wait for child thread*/
   pthread_join(tid,NULL);
}
void *runner(void *param){
   pthread_t tid2 = syscall(SYS_gettid);
   int i, upper = atoi(param);  sum = 0;
   for (i = 1; i <= upper; i++)
        sum += i;
   pthread_exit(0);
}
```

```
Ps -T
PID   SPID                          CMD
4222  4222   pts/0   00:00:00   texit1
4222  4223   pts/0   00:00:00   texit1
4224  4224   pts/0   00:00:00    ps
Child thread exit, ID is 4223
ps -T
PID   SPID                          CMD
4222  4222    pts/0   00:00:00   texit1
4225  4225    pts/0   00:00:00   ps
```

Terminating thread: pthread_detach

- The pthread_detach function sets a thread's internal options to specify that storage for the thread can be reclaimed when the thread exits.

```
int pthread_detach(pthread_t thread);
pthread_detach(pthread_self());
pthread_exit();
```

- Here the thread detach itself. Here is another example

```
pthread_create(&tid, NULL, processfd, &fd);
pthread_detach(tid);
```

- Essentially, threads that are detached release all their resources once they exit.

- To prevent memory leaks, long-running programs should eventually call either pthread_detach or pthread_join for every thread.

Terminating process

- A process can terminate by calling `exit` directly, by executing `return` from `main`, or by having one of the other process threads call `exit`. (see texit2.c)

- In any of these cases, all threads terminate. If the main thread has no work to do after creating other threads, it should either block until all threads have completed or call `pthread_exit(NULL)`.

- A call to `exit` causes the entire process to terminate; a call to `pthread_exit` causes only the calling thread to terminate

- A process will exit if its last thread calls `pthread_exit`.

texit2.c

```
int sum;void *runner(void *param);
int main(int argc, char *argv[]) {
   pthread_t tid;
   pthread_create(&tid, &attr, runner, argv[1]);
   sleep(10);  /*wait for child thread*/
   pthread_join(tid,NULL);
   pthread_exit(0);
}
void *runner(void *param){
   pthread_t tid2 = syscall(SYS_gettid);
   int i, upper = atoi(param);  sum = 0;
   for (i = 1; i <= upper; i++)
        sum += i;
   exit(0);
}
```

```
ps-T
  PID     SPID                              CMD
4402    4402    pts/0    00:00:00    texit2
4402    4403    pts/0    00:00:00    texit2
4404    4404    pts/0    00:00:00     ps
Child thread exit with exit(0), ID is 4403
ps -T
PID    SPID                                 CMD
4405  4405     pts/0    00:00:00    ps
```

Killing a thread

- Threads can force other threads to return through the cancellation mechanism.
- A thread calls `pthread_cancel` to request that another thread be canceled.
- The single parameter of `pthread_cancel` is the thread ID of the target thread to be canceled.
- The `pthread_cancel` function does not cause the caller to block while the cancellation completes. Rather, `pthread_cancel` returns after making the cancellation request.
- (see texit3.c)

texit3.c

```
int sum; void *runner(void *param);
int main(int argc, char *argv[]){
   pthread_t tid;
   pthread_create(&tid, &attr, runner, argv[1]);
   sleep(4);
   pthread_cancel(tid);
   pthread_join(tid,NULL);
   printf("Parent thread has canceled the child thread \n");
   pthread_exit(0);}
}
void *runner(void *param){
   int i, upper = atoi(param);
   sleep(10);
   pthread_exit(0);
}
```

```
ps-T
 PID    SPID                      CMD
4478   4478   pts/0   00:00:00   texit3
4478   4479   pts/0   00:00:00   texit3
4480   4480   pts/0   00:00:00    ps
Parent thread has canceled the child thread
ps -T
PID    SPID                       CMD
4478   4478   pts/0   00:00:00   texit3
4481   4481   pts/0   00:00:00   ps
```

Same thread program for Windows

```
#include <Windows.h> #include <stdio.h>
DWORD Sum;
DWORD WINAPI Summation(LPVOID Param){
  DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
       Sum += i;
   return 0;
}
int main(int argc, char *argv[] ){
                DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   Param = atoi(argv[1]);
   ThreadHandle = CreateThread(NULL,0,Summation,&Param,0,&ThreadId) ;
                if(ThreadHandle != NULL){
       WaitForSingleObject(ThreadHandle, INFINITE);
       CloseHandle(ThreadHandle);
       printf("sum= %d\n", Sum);
   }
}
```

(see C:Teaching/HUST/Operating Systems/Exercises/TW)

Two other examples

- See pdf files PthreadEx1 and PthreadEx2 also the C code for these example in thread2.c and thread3.c
- PthreadEx1 has 3 threads, the main and 2 others.
  - The two threads execute the same function
  - Two data structures are declared and initialized in main, one is passed as argument to thread1 and the other to thread2.
  - Each thread print the content of the received data structure.
- PthreadEx2 defines an array of 1000000 double.
  - The program adds number from 1 to 1000000, this addition is parallelize using some number of threads.
  - Each thread adds a different sub-sequence of 1000000/4 numbers.
  - Once a thread has completed the addition of its sub-sequence of numbers it write the value into a global variable sum.
  - To avoid that two threads write in same time in the global variable, a synchronization primitive is used to guarantee exclusive access to the global variable.

thread2.c

```
void *message_function ( void *ptr );
typedef struct str_thdata{
   int thread_no;  char message[100];
} thdata;   /* structs to be passed to threads */
int main(){
    pthread_t thread1, thread2;
    thdata data1, data2;
    data1.thread_no = 1;
    strcpy(data1.message, "Hi prof!");
    data2.thread_no = 2;
    strcpy(data2.message, "Hi Students!");
    pthread_create (&thread1, NULL, message_function, (void *) &data1);
    pthread_create (&thread2, NULL, message_function, (void *) &data2);
    pthread_join(thread2, NULL);
    pthread_join(thread1, NULL);
    pthread_exit(0);
}

void *message_function ( void *ptr ){
     thdata *data;
     data = (thdata *) ptr;
     printf("Thread %d with thread id %ld says %s \n", data->thread_no, tid1,  data->message);
      sleep(5);
      pthread_exit(0);
}
```

thread3.c: child thread

```
#define NTHREADS     4
#define ARRAYSIZE   1000000
#define ITERATIONS   ARRAYSIZE / NTHREADS
double  sum=0.0, a[ARRAYSIZE];
pthread_mutex_t sum_mutex;
void *do_work(void *tid) {
    int i, start, *mytid, end;
    double mysum=0.0;
    mytid = (int *) tid;
    start = (*mytid * ITERATIONS);
    end = start + ITERATIONS;
    for (i=start; i < end ; i++) {a[i] = i * 1.0;   mysum = mysum + a[i];   }
    pthread_mutex_lock (&sum_mutex);
    sum = sum + mysum;
    pthread_mutex_unlock (&sum_mutex);
    sleep(15);  pthread_exit(NULL);
}
```

thread3.c

```
#define NTHREADS     4
#define ARRAYSIZE   1000000
#define ITERATIONS   ARRAYSIZE / NTHREADS
double  sum=0.0, a[ARRAYSIZE];
pthread_mutex_t sum_mutex;
int main(int argc, char *argv[]){
     int i, start, tids[NTHREADS];
     pthread_t threads[NTHREADS];
     pthread_mutex_init(&sum_mutex, NULL);
     for (i=0; i<NTHREADS; i++) {
          tids[i] = i;
          pthread_create(&threads[i], &attr, do_work, (void *) &tids[i]);
     }
     /* Wait for all threads to complete then print global sum */
     for (i=0; i<NTHREADS; i++)
           pthread_join(threads[i], NULL);
     printf ("Done. Sum= %e \n", sum);
     sum=0.0;
     for (i=0;i<ARRAYSIZE;i++){
          a[i] = i*1.0;  sum = sum + a[i]; }
     printf("Check Sum= %e\n",sum);  /* Clean up and exit */
     pthread_mutex_destroy(&sum_mutex);
     pthread_exit (NULL);
}
```

Why use threads

- There are four major categories of benefits to multi-threading:
    1. Responsiveness:
        - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
    2. Resource sharing:
        - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
    3. Economy:
        - Creating and managing threads (and context switches between them) is much faster than performing the same for processes.
    4. Parallelism, i.e. utilization of multiprocessor architectures:
        - A single threaded process can only run on one CPU, whereas the execution of a multi-threaded application may be split amongst available processors.

Parallelism vs concurrency

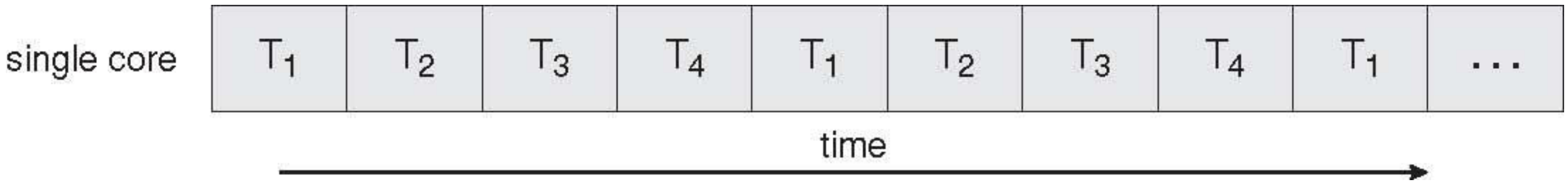
- Parallelism

  implies a system can perform more than one task simultaneously

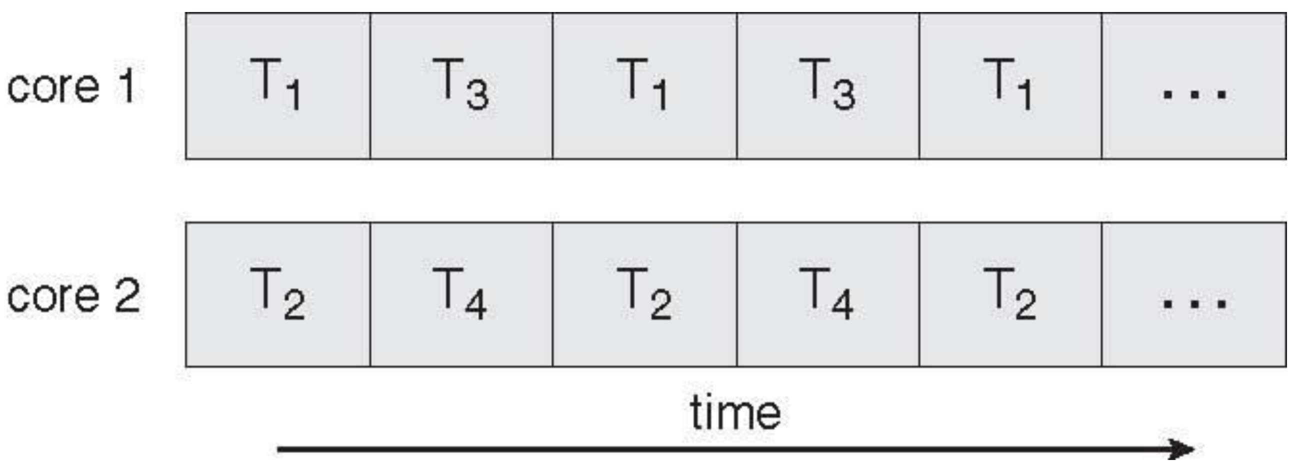
- Concurrency
  - supports more than one task making progress

Single/Multicore systems

- In a single core, multithreading only implement concurrency where thread interleave over time but only one thread is executed at a time



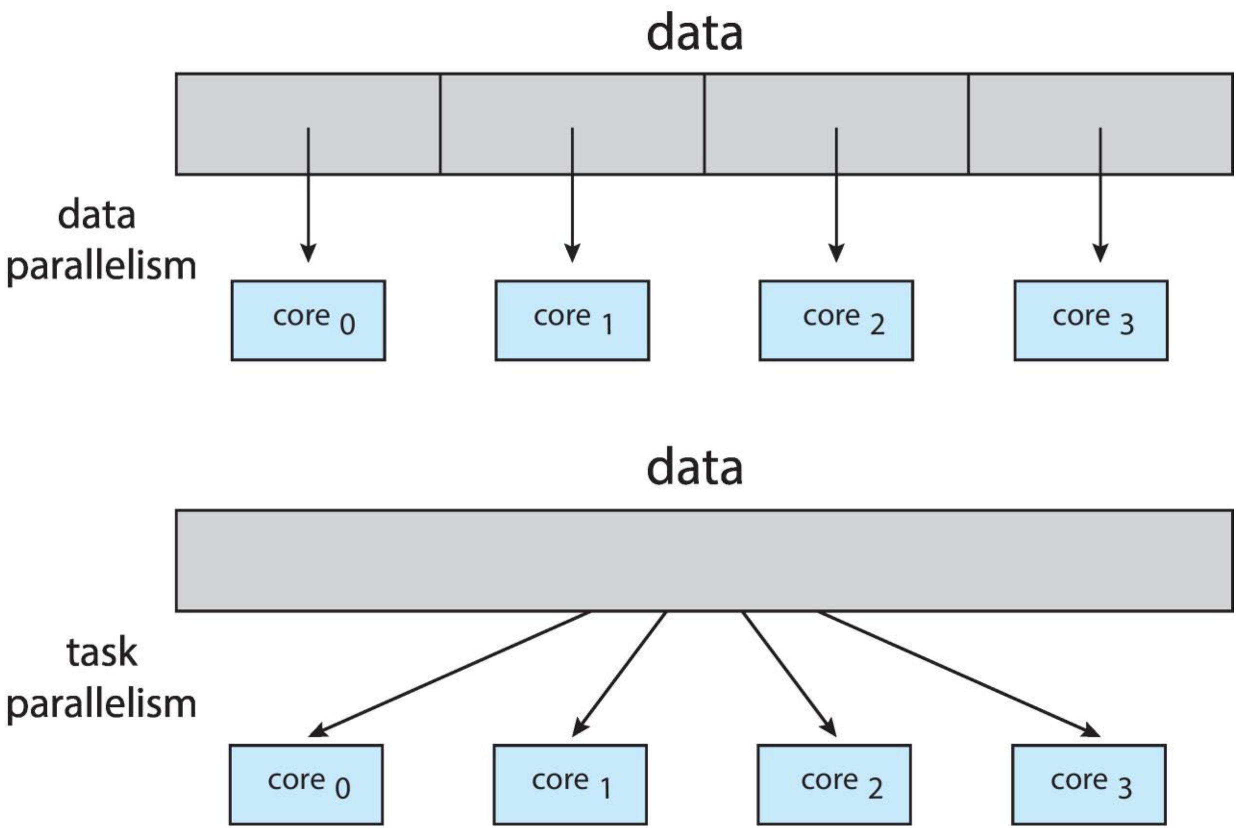
- Multiple computing cores on a single cheap
- Multithreading helps to efficiently used multiple core architectures, threads may run in parallel



Multicore Programming

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

Data and Task Parallelism



Threads attributes: State

```
pthread_attr_t attr; /* set of thread attributes */
pthread_attr_init(&attr);
pthread_create(&tid, &attr, runner, argv[1]);
```

- When a thread is created, it is passed a set of attributes which are used to run the thread.
- Settable properties of thread attributes are state, stack, scheduling.
- The pthread_attr_init function initializes a thread attribute with default values. This is the case with the example above.
- Thread state: The possible values of the thread state are PTHREAD_CREATE_JOINABLE and PTHREAD_CREATE_DETACHED.
- By default, threads are joinable. You can detach a thread by calling the pthread_detach function after creating the thread. Alternatively, you can create a thread in the detached state by using an attribute object with thread state PTHREAD_CREATE_DETACHED

Threads attributes: Stack

- A thread has a stack whose location and size are user-settable. To define the placement and size of the stack for a thread, you must first create the stack attributes using `pthread_attr_setstack` function which sets the stack parameters

- First the function `pthread_attr_getstack` must be called to examine the stack parameters

Threads attributes: Scheduling

- The *contention scope* controls whether the thread competes within the process or at the system level for scheduling resources.
- The pthread_attr_getscope examines the contention scope, and the pthread_attr_setscope sets the contention scope of the thread
- The following code segment creates a thread that contends for kernel resources:

```
pthread_attr_t tattr;
pthread_t tid;
pthread_attr_init(&tattr);
pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
pthread_create(&tid, &tattr, processfd, &fd);
```

Threads attributes: Scheduling

- Thread scheduling also specifies the priority of the thread

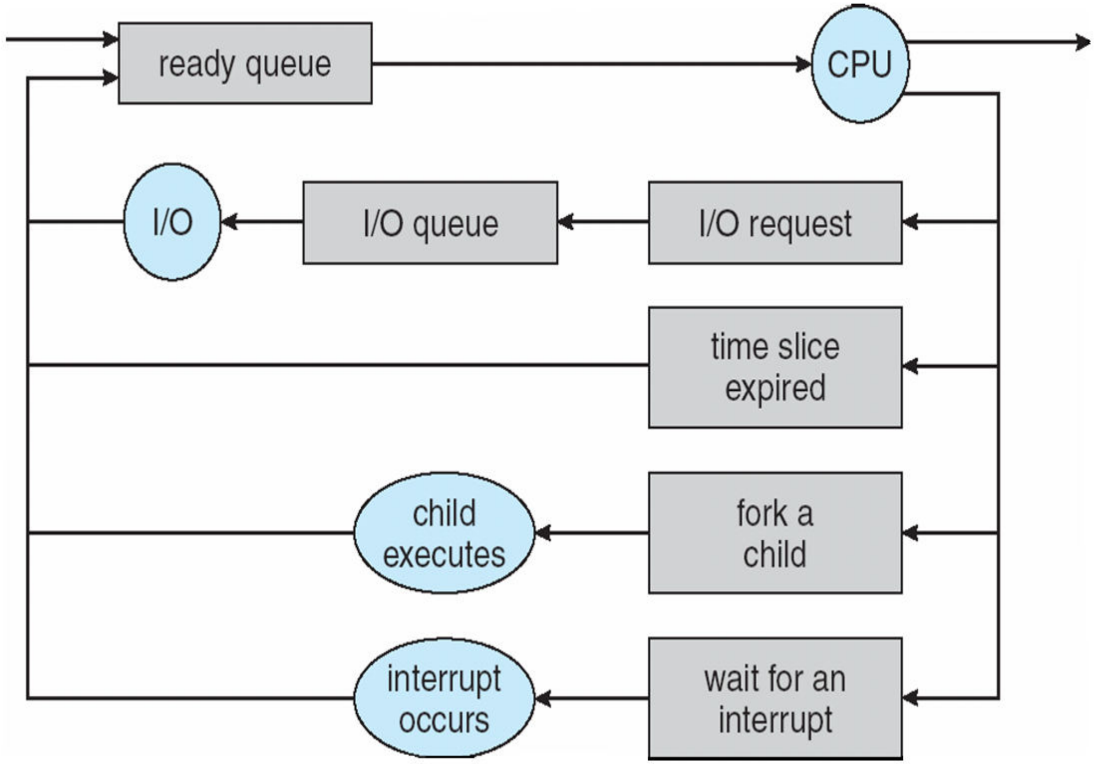
- The sched_priority field holds an int priority value, with larger priority values corresponding to higher priorities. Implementations must support at least 32 priorities
- The following code segment creates a dothis thread with the default attributes, except that the priority is HIGHPRIORITY.

```
int fd;
pthread_attr_t tattr;
pthread_t tid;
struct sched_param tparam;
tattr = makepriority(HIGHPRIORITY);
pthread_create(&tid, tattr, dothis, &fd);
```

- Threads of the same priority compete for processor resources as specified by their scheduling policy. The sched.h header file defines SCHED_FIFO for first-in-first-out scheduling, SCHED_RR for round-robin scheduling and SCHED_OTHER for some other policy

Scheduling

- When processes support multi-threading the scheduling and execution is done at the thread level



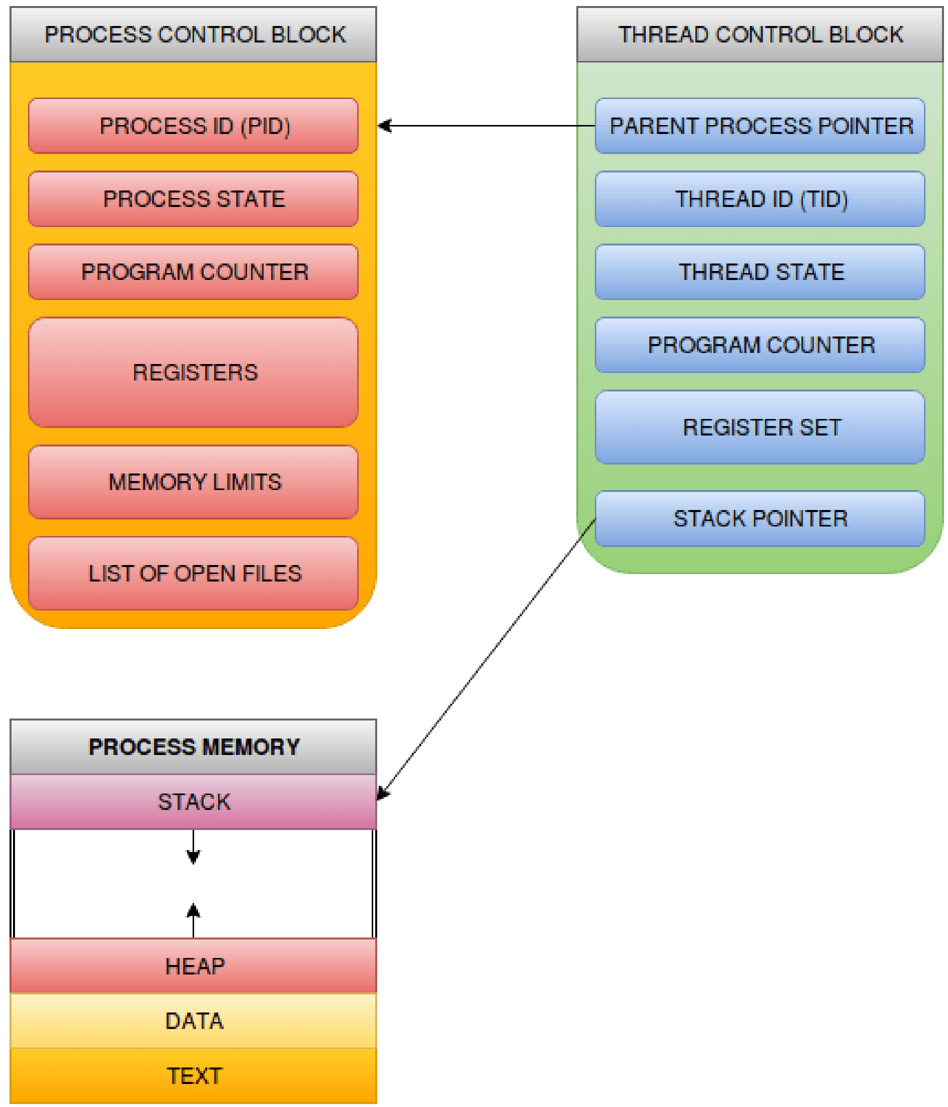
Thread control block (TCB)

- Very similar to Process Control Block (PCB) which represents processes, **Thread Control Blocks (TCBs)** represents threads generated in the system
- The TCB includes:
  - Thread Identifier: Unique id (tid) is assigned to every new thread
  - Stack pointer: Points to thread's stack in the process
  - Program counter: Points to the current program instruction of the thread
  - State of the thread: (running, ready, waiting, start, done)
  - Thread's register values:
  - Pointer to the process control block (PCB) of the process that the thread lives on

Thread control block

Thread ID
Thread state
CPU information : Program counter Register contents
Thread priority
Pointer to process that created this thread
Pointer(s) to other thread(s) that were created by this thread

Process vs thread control blocks



Process Context Switch

- The steps in a process context switch are:
  1. Save context of processor including program counter and other registers
  2. Update the process control block of the process that is currently in the Running state
  3. Move process control block to appropriate queue – ready; blocked; ready/suspend
  4. Select another process for execution
  5. Update the process control block of the process selected
  6. Update memory-management data structures
  7. Restore context of the selected process

- `vmstat 1 3` The first line gives the average number of context switches over 1 second since the system booted, and the next two lines give the number of context switches over two 1-second intervals
- cat /proc/2166/status number of context switches for a given process

Thread context switch

A thread context switch can be understood as a context switch inside a process:

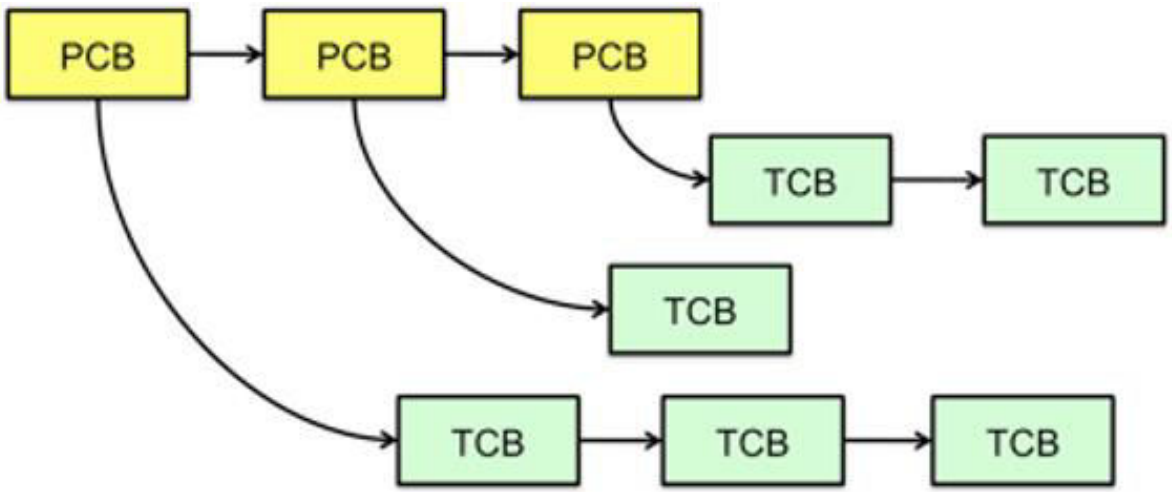
1-Save context of processor including program counter and other registers in TCB

2-Update the thread control block of the thread that is currently in the Running state

3- Move thread control block to appropriate queue – ready; blocked; ready/suspend

4- Select another thread for execution

5- Update the thread control block of the thread selected



Threads context switch within different processes

1.Process and thread context switch are the same up to this point:

6- the kernel checks if the scheduled and unscheduled threads belong to the same process.

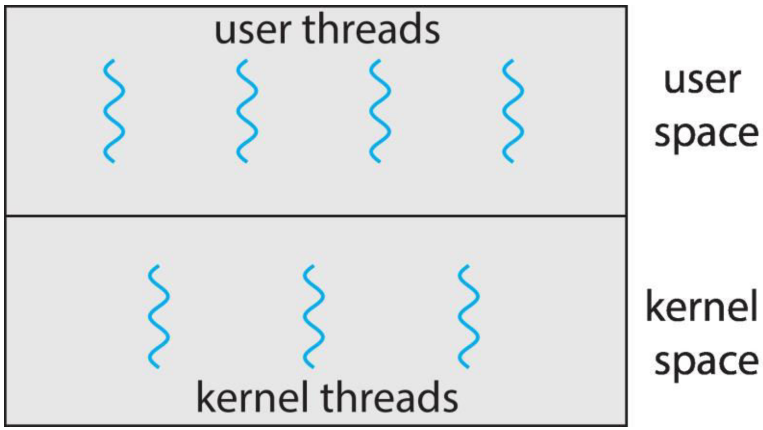
7- If not ("process" rather than "thread" switch), the kernel resets the current address space by pointing the MMU (Memory Management Unit) to the page table of the scheduled process.

8- The TLB (Translation Lookaside Buffer), which is a cache containing recent virtual to physical address translations, is also flushed to prevent erroneous address translation.

Note that these are the only steps in the entire set of context switch actions that cares about processes!

User and Kernel Threads

- Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**.
- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.
- However there must be a relationship between user threads and kernel threads



User level Threads

- For user level threads, the thread library is implemented entirely in user space with no kernel support.

- All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

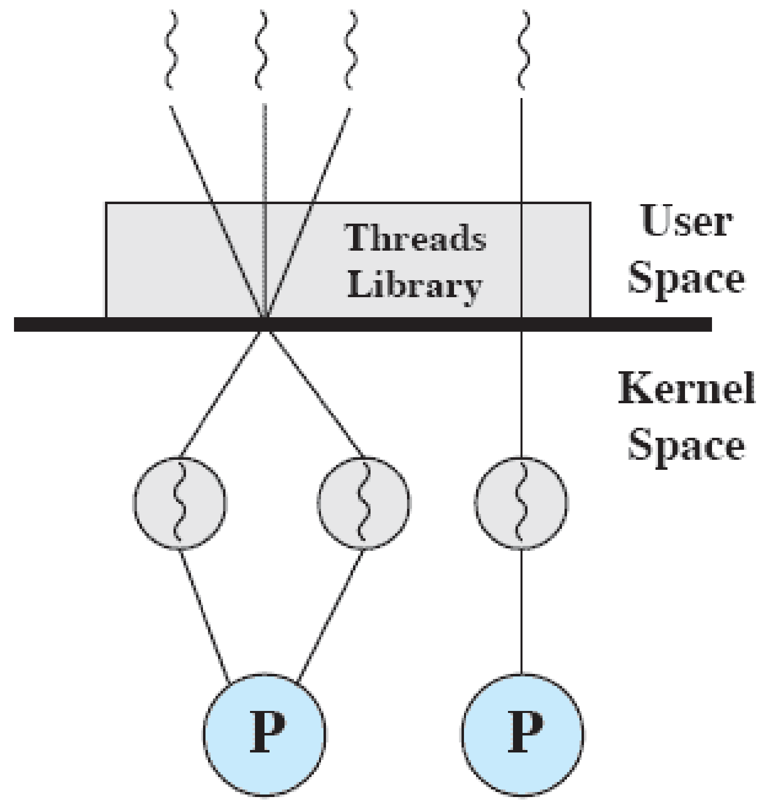
- User-level threads have low overhead, but they have the disadvantages of running as a single thread at the kernel level, therefore no parallel execution is possible among the user threads of a same process.

Kernel Threads (KLT)

- For kernel threads, the thread library is implemented at the kernel-level library and supported directly by the operating system.
- In this case, code and data structures for the library exist in kernel space.
- Invoking a function in the API for the library typically results in a system call to the kernel, for example creating a thread results in a system call
- Virtually all general -purpose operating systems support kernel threads, including:
  - Windows, Linux, Mac OS X, iOS, Android
- Kernel-level threads have more overhead (can be almost as expensive as the scheduling of processes themselves) but kernel level threads of a same process can execute in parallel

Mapping user threads to kernel threads

- The user writes the program in terms of user-level threads and then specifies how many kernel-schedulable entities are associated with the process.
- The user-level threads are mapped into the kernel-schedulable entities at runtime to achieve parallelism.
- Linux and Windows OS's map each user level thread to a kernel thread, mappings is one-to-one



(c) Combined

POSIX threads mapping

- POSIX uses *thread-scheduling contention scope*, which gives the programmer some control over how user level threads are mapped to kernel threads.

- A thread can have a `contentionscope` attribute of either `PTHREAD_SCOPE_PROCESS` or `PTHREAD_SCOPE_SYSTEM`.

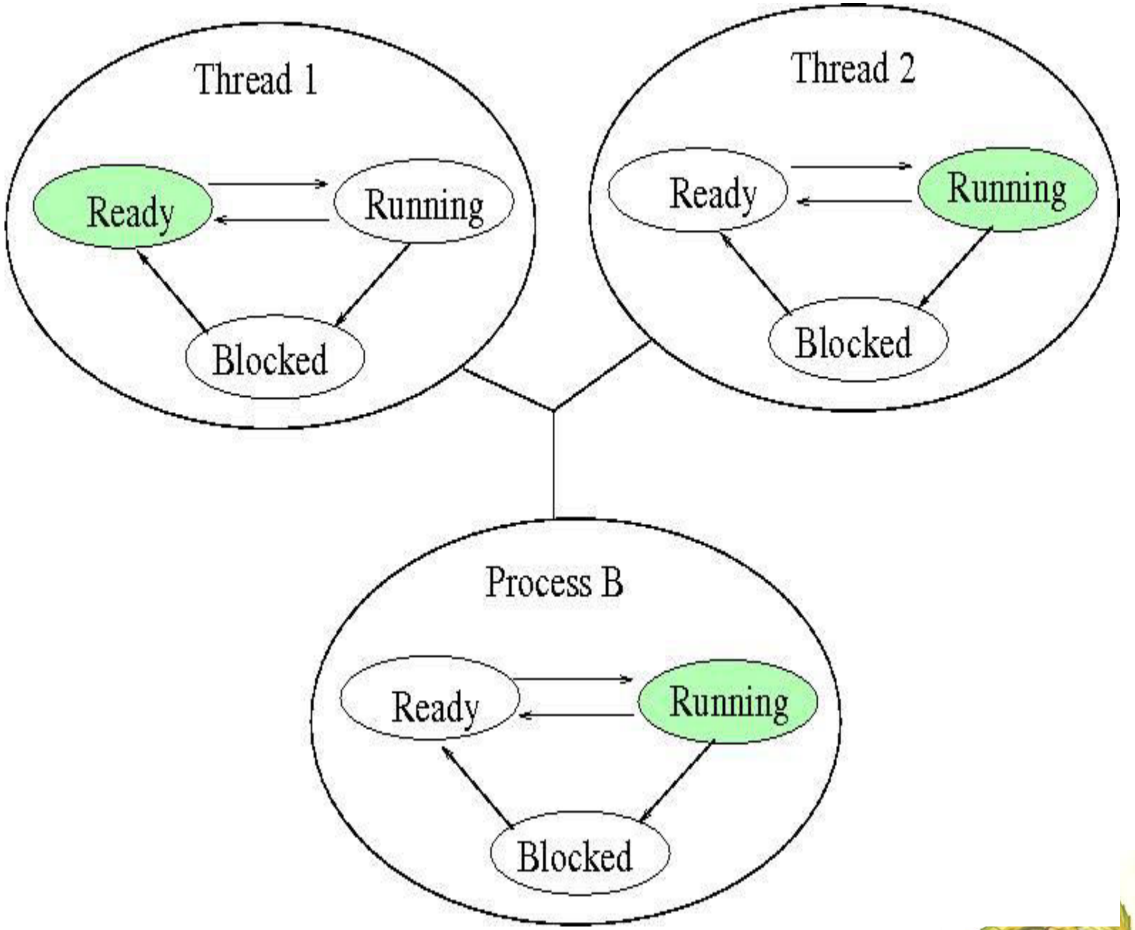
- Threads with the `PTHREAD_SCOPE_PROCESS` attribute contend for processor resources with the other threads in their process.

- Threads with the `PTHREAD_SCOPE_SYSTEM` attribute contend systemwide for processor resources, much like kernel-level threads.

- POSIX leaves the mapping between `PTHREAD_SCOPE_SYSTEM` threads and kernel entities up to the implementation, but the obvious mapping is to bind such a thread directly to a kernel entity.

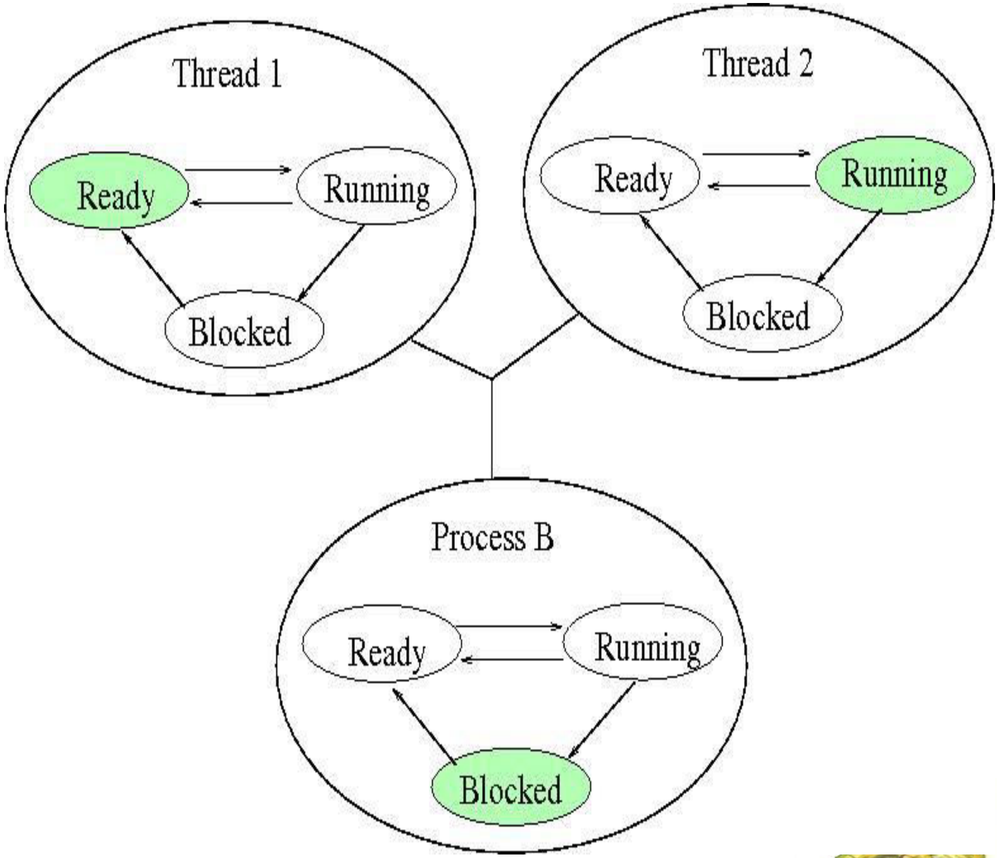
Many-to-one: many ULTs to one KLT

- Process B is running
- Thread one is ready
- Thread 2 is running



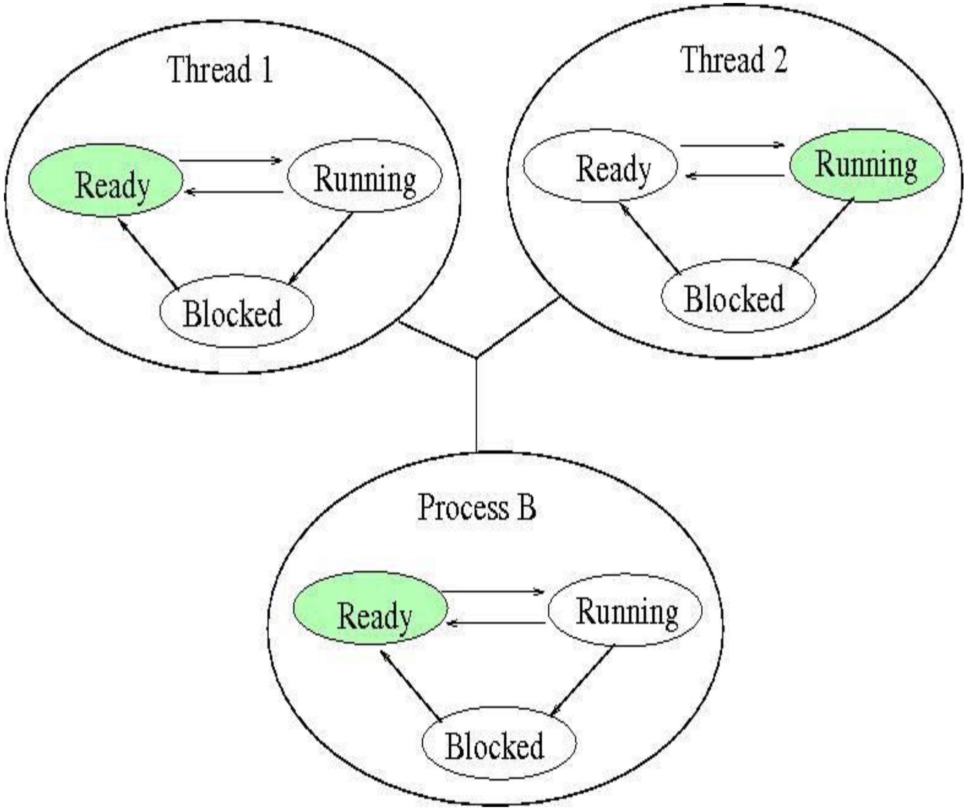
Many ULTs versus one KLT (2)

- Thread 2 made a system call
- Which blocks process B



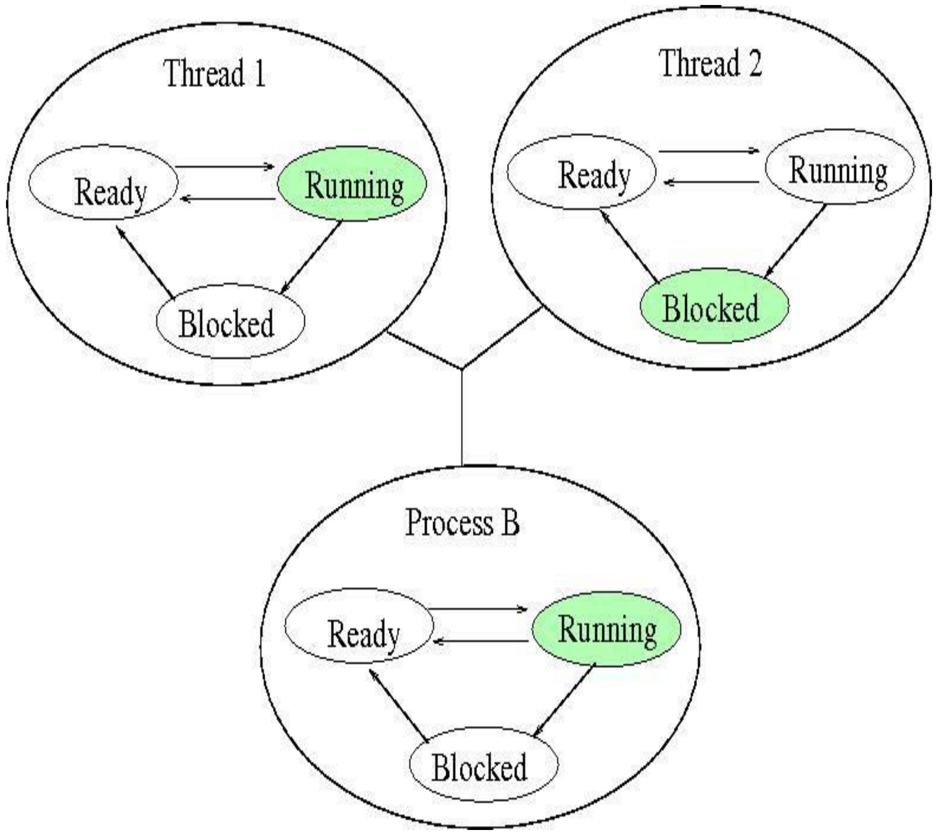
Many ULTs versus one KLT (3)

- Once the system call has been served, OS place process B in the Ready queue
- According to data structure of the user level threads library, thread 2 still running



Many ULTs versus one KLT (4)

- Process B moves into the running state
- Thread 2 needs actions performed by thread 1
- Thread 2 enters in a blocked state
- Thread 1 transits from Ready to Running



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread execution
- Thread cancellation of target thread

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
  - Some Unix (Solaris) have two versions of fork():
    - One duplicate all threads: forkall()
    - One duplicate only the thread invoking fork1()
- The execution of exec() destroy completely the calling program, including all the threads belonging to the corresponding process
  - Should not use the fork that duplicate all threads if exec() is called

Relations Threads/Processes

- Actions that affect all of the threads in a process:
  - The OS must manage these at the process level
- Examples:
  - Suspending a process (swapping the process out of the main memory) involves suspending all threads of the process
  - Termination of a process, terminates all threads within the process

Thread Execution States

- Key thread states are Running, Ready and Waiting
- No suspend states as it is a per-process concept
  - If a process is swapped out, so all the threads since they shared the same address space with the swapped process

Thread Execution States (cont)

- A thread is blocked when waiting for an event (saving its registers, program counter and stack pointers)
- The occurrence of the event on which the thread was blocked triggers the thread to be placed in the ready queue
- A thread within the process may spawn other threads
- May synchronize with other threads
  - Similar to processes

Operating System Examples

- Linux Threads

Linux Process/Thread Model

- Linux does not recognize a distinction between processes and threads
  - it uses the more generic term "tasks".
- Linux has a single command to create processes or threads, it is **clone()**
- clone( ) allows for varying degrees of sharing between the parent and child tasks, controlled by flags such as those shown in the following table:

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.

- Calling clone( ) with no flags set is equivalent to fork( ).
- Calling clone( ) with CLONE_FS, CLONE_VM, CLONE_SIGHAND, and CLONE_FILES is equivalent to creating a thread, as all of these data structures will be shared.

Clone(): example

- The flag CLONE_VFORK means the execution of the calling process is suspended until the child exit
- The flag CLONE_VM has been defined, if not used, the two tasks are in different memory spaces, the parent task is copied in the child task, any changes by the child task will not be visible to the parent

```
#define _GNU_SOURCE #include <stdio.h> #include <unistd.h> #include <stdlib.h>
#include <sched.h> #include <signal.h>
#define FIBER_STACK 8192
void * stack; int b = 0;
int do_something(){
   while (b<10){
      printf("pid : %d, b = %d\n", getpid(), b++);
   }
   exit(0);
}
int main() {
   void * stack;
   stack = malloc(FIBER_STACK);
   int a = 0;
   if (a == 0)
      clone(&do_something, (char *)stack + FIBER_STACK, CLONE_VM|CLONE_VFORK, 0);
      //clone(&do_something, (char *)stack + FIBER_STACK, CLONE_VFORK, 0);
   while (a<10){
      printf("pid : %d, a = %d, b = %d\n", getpid(), a++,b);
   }
   free(stack);
   exit(0);
}
```

End of Section 4