Topics: Threads
(SGG, 4.1-4.4)
(USP, Chapter 12 Pthreads)

CS 3733 Operating Systems

Instructor: Dr. Turgay Korkmaz

Department Computer Science

The University of Texas at San Antonio

Office:	**NPB 3.330**
Phone:	**(210) 458-7346**
Fax:	**(210) 458-4437**
e-mail:	**korkmaz@cs.utsa.edu**
web:	**www.cs.utsa.edu/~korkmaz**

These slides are prepared based on the materials provided by the textbook [**SGG**] .

Chapter 4: Threads

- Overview *
- Multithreading Models *****
- Thread Libraries ****
  - **Pthreads** and **Java threads**
- Threading Issues ***
- Operating System Examples *
- Windows XP Threads *
- Linux Threads *

Objectives

- To introduce the notion of a thread
  - a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
  - Identify the basic components of a thread, and contrast threads and processes

https://www.youtube.com/watch?v=M9HHWFp84f0

https://www.youtube.com/watch?v=5sw9XJokAqw

- Describe the benefits and challenges of designing multithreaded applications
- To examine issues related to multithreaded programming
- To discuss the APIs for the **Pthreads** and **Java threads** libraries

Motivation: Example 1: Monitoring multiple file FDs?

- Methods you may know about (USP book):
  - Separate processes
    - each monitors one file descriptor (Program 4.11)
  - A single process with
    - select (Program 4.12 and Program 4.14)
    - poll (Program 4.17)
    - Nonblocking I/O with polling (Example 4.39)
    - POSIX asynchronous I/O (Program 8.14 and Program 8.16)
- A New approach: THREADS
  - A single process with separate threads
    - each monitors one file descriptor (Section 12.2)

A program that monitors two files by forking a child process (Program 4.11)

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "restart.h"
```

```
int main(int argc, char *argv[]) {
  int bytesread;
  int childpid;
  int fd, fd1, fd2;
  if (argc != 3) {
    fprintf(stderr, "Usage: %s file1 file2\n", argv[0]);
    return 1;
  }
  if ((fd1 = open(argv[1], O_RDONLY)) == -1) {
    fprintf(stderr, "Failed to open file %s:%s\n", argv[1], strerror(errno));
    return 1;
  }
  if ((fd2 = open(argv[2], O_RDONLY)) == -1) {
    fprintf(stderr, "Failed to open file %s:%s\n", argv[2], strerror(errno));
    return 1;
  }
  if ((childpid = fork()) == -1) {
    perror("Failed to create child process");
    return 1;
  }
  if (childpid > 0) /* parent code */
     fd = fd1;
  else
     fd = fd2;
  bytesread = copyfile(fd, STDOUT_FILENO);
  fprintf(stderr, "Bytes read: %d\n", bytesread);
  return 0;
}
```

\+ Easy to program

\- Sharing data between processes is hard

Single process with select:

A function that blocks until one of two file descriptors is ready

```
#include <errno.h>
#include <string.h>
#include <sys/select.h>

int whichisready(int fd1, int fd2) {
  int maxfd;
  int nfds;
  fd_set readset;
  if ((fd1 < 0) || (fd1 >= FD_SETSIZE) ||
      (fd2 < 0) || (fd2 >= FD_SETSIZE)) {
    errno = EINVAL;
    return -1;
  }
  maxfd = (fd1 > fd2) ? fd1 : fd2;
  FD_ZERO(&readset);
  FD_SET(fd1, &readset);
  FD_SET(fd2, &readset);
  nfds = select(maxfd+1, &readset, NULL, NULL, NULL);
  if (nfds == -1)
     return -1;
  if (FD_ISSET(fd1, &readset))
     return fd1;
  if (FD_ISSET(fd2, &readset))
     return fd2;
  errno = EINVAL;
  return -1;
}
```

- Coding is getting complicated.
+ Sharing data is easy

A single process with separate Threads for monitoring two FDs.

- When creating a thread, you indicate which C function the thread should execute.
- When a new thread is created, it runs concurrently with the creating process/thread. This contrasts with function call!

So, what is the benefit?

+ Easy to program
+ Easy to share data

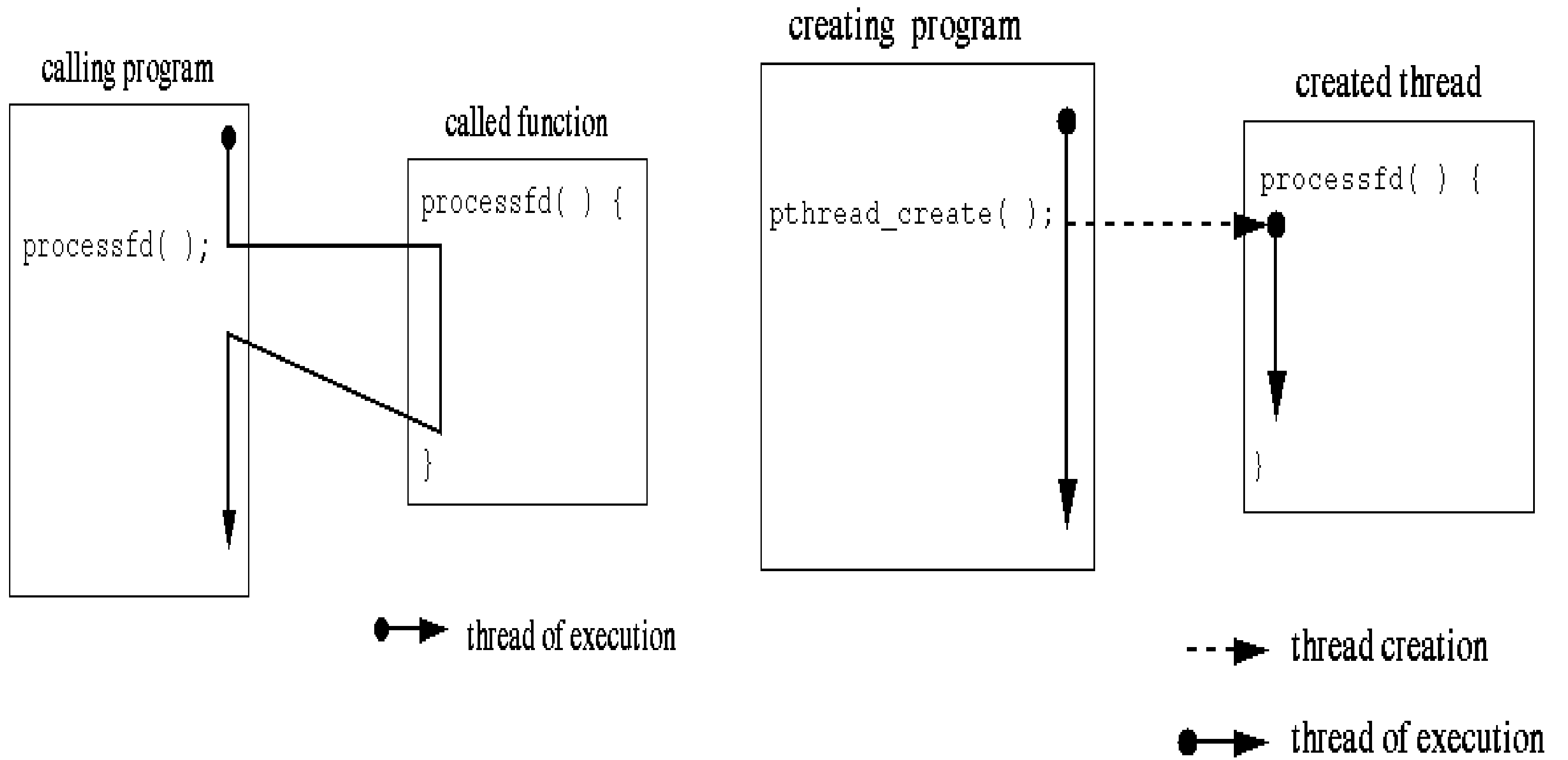
```
#define BUFSIZE 1024
void *processfd(void *arg);
main(){
  int error;
  int fd1, fd2;
  pthread_t tid1, tid2;
  /* open files… */
  if (error = pthread_create(&tid1, NULL, processfd, &fd1))
     fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
  if (error = pthread_create(&tid2, NULL, processfd, &fd2))
     fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
}

void docommand(char *cmd, int cmdsize);
void *processfd(void *arg) { /* process commands read from file descriptor */
  char buf[BUFSIZE];
  int fd;
  ssize_t nbytes;
  fd = *((int *)(arg));
  for ( ; ; ) {
    if ((nbytes = r_read(fd, buf, BUFSIZE)) <= 0)
      break;
    docommand(buf, nbytes);
  }
  return NULL;
}
```

- A function that is used as a thread must have a special format.
- It takes a single parameter of type **pointer to void** and returns a pointer to void.
- The parameter type allows any pointer to be passed.
- This can point to a structure, so in effect, the function can use any number of parameters.

****

Function Call vs. Thread

```
processfd(&fd1);  // vs.
        pthread_create(&tid1, NULL, processfd, &fd1));
```



a single thread of execution

a separate thread of execution for function

Then the benefits are ….

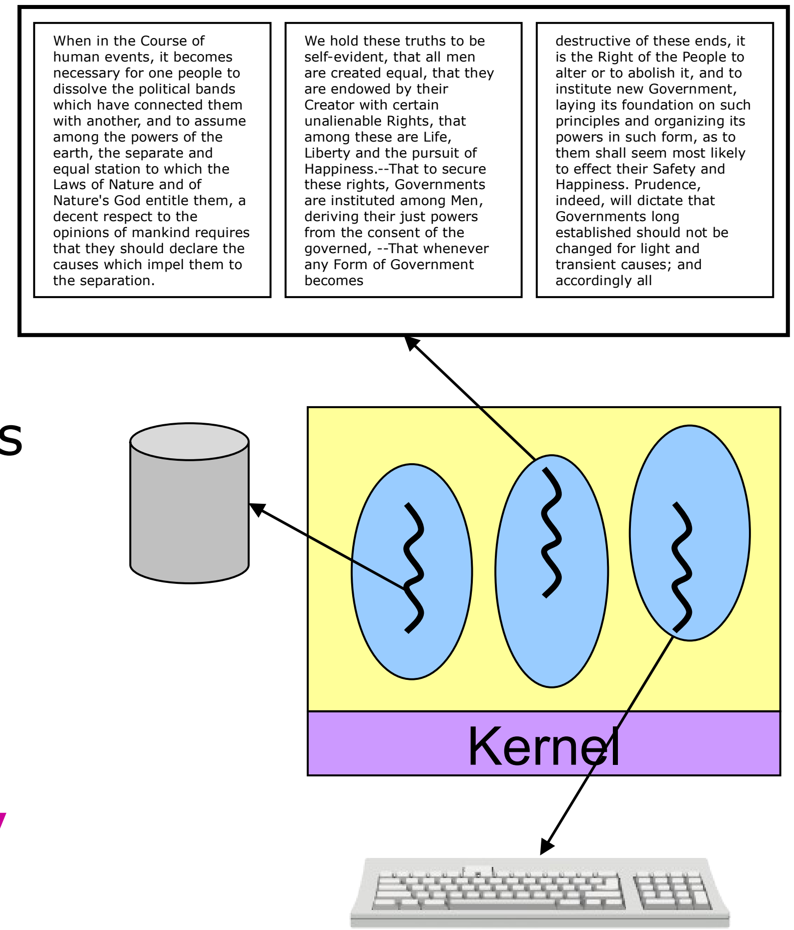
Running **multiple** tasks/threads **concurrently**

Sharing data between threads is easy (like a single process)
while
coding is also easy (like multiple processes)…

Best of the both worlds…

Example 2: A Multi-Activity Text Editor

- **Process approach** on data
  - P1: read from keyboard
  - P2: format document
  - P3: write to disk



The processes will *actively* access the **same set of data.**

How do the processes exchange data?

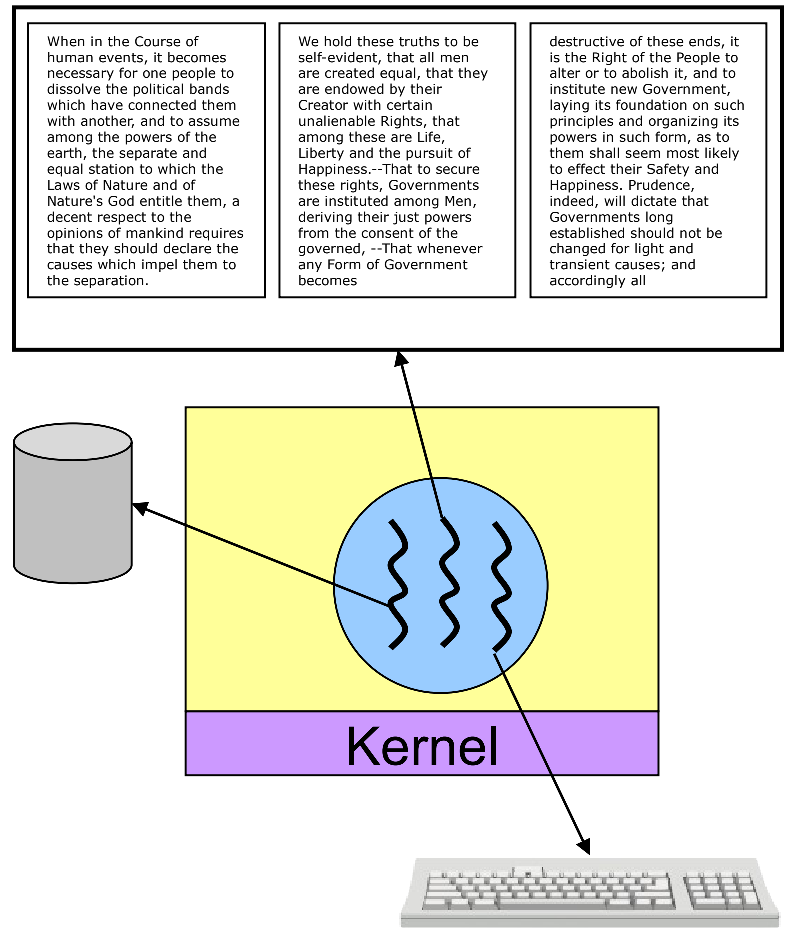
Context Switch for Processes- **costly**

Ideal Solution for the Text Editor

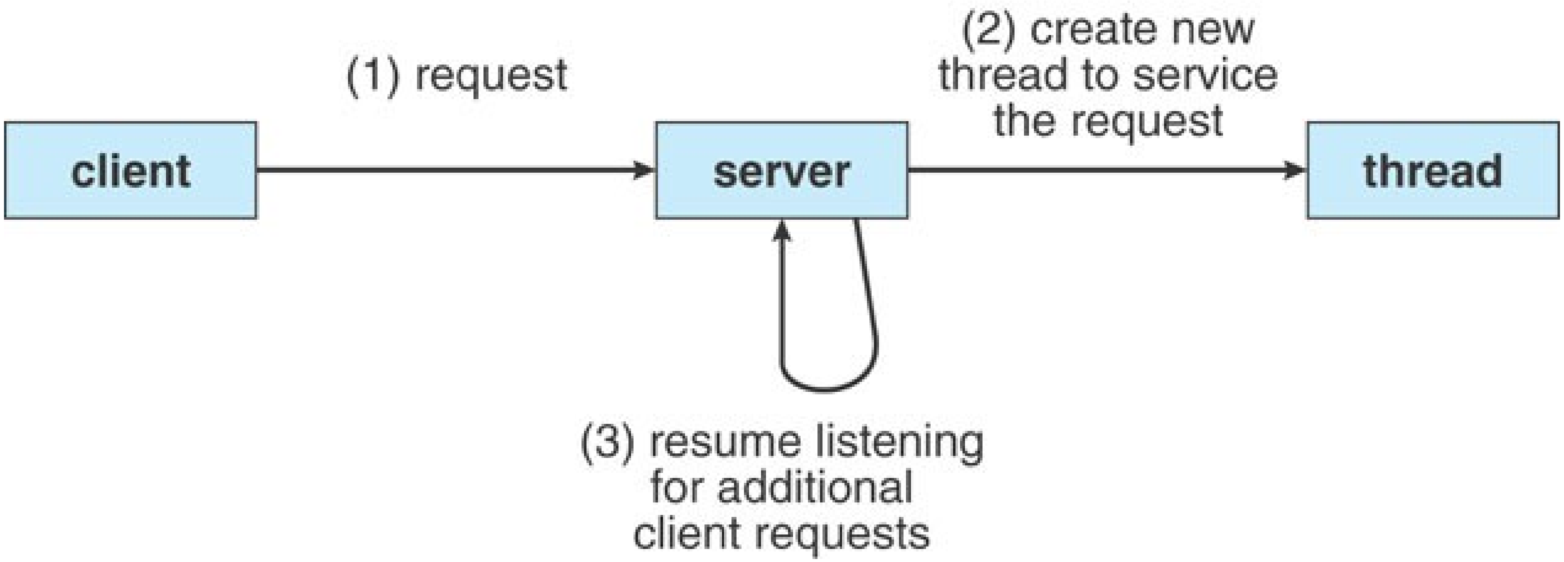
Threads

- Three *activities* within one process
  - Single address space
  - Same execution environment
  - Data shared easily

- Switch between activities
  - Only *running context*
  - ***No change in address space***



Another Example: Web servers



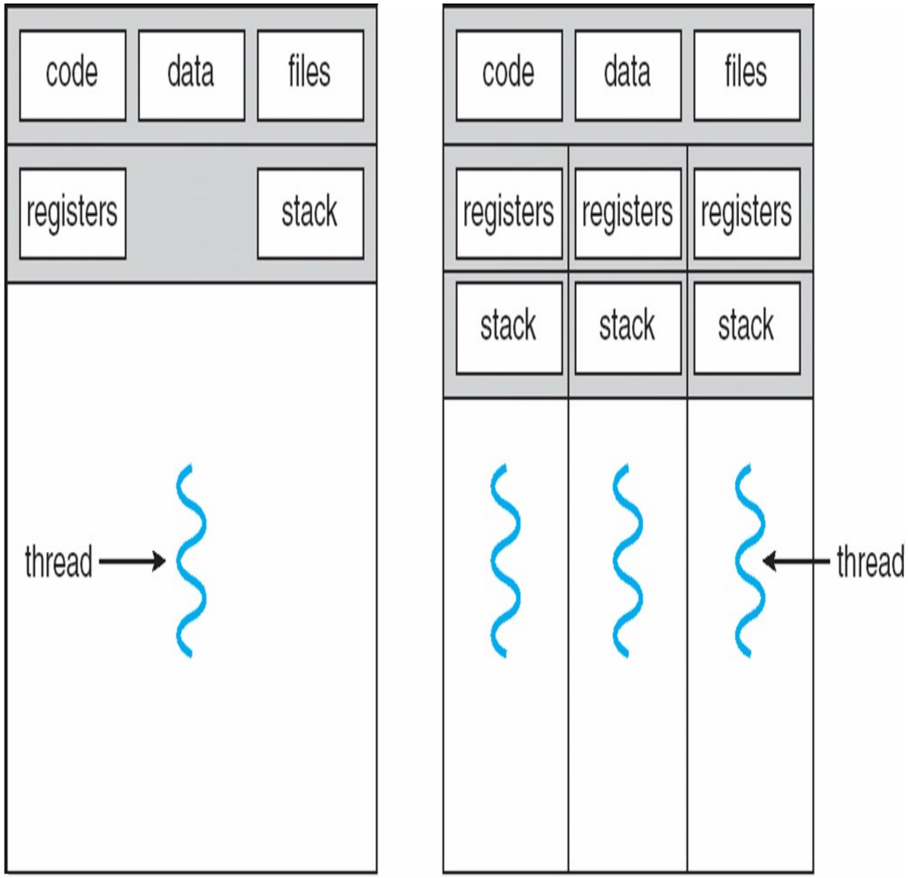
Recap: Motivation

- 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

Thread vs. Process

Both are methods for Concurrency and Parallelism

- Processes
  - Independent execution units use their own states, address spaces, and interact with each other via IPC
  - Traditional Processes have single flow of control (thread)
- Thread
  - Flow of control (activity) within a process
  - A single process on a modern OS may have *multiple* threads
  - All threads share the code, data, and files while having some separated resources
  - Threads directly interact with each other using shared resources



single-threaded process

multithreaded process

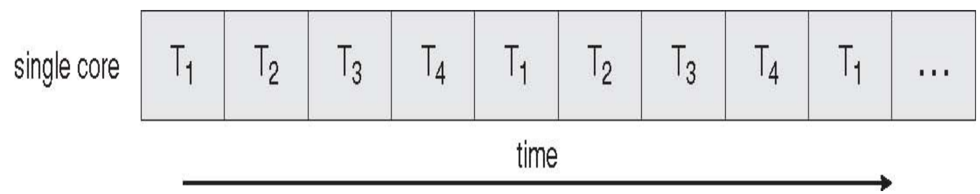
Concurrency vs. Parallelism?

Benefits of multithreaded programming

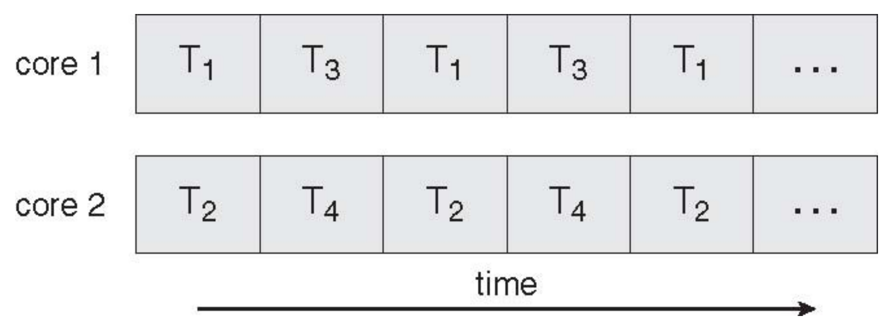
- **Responsiveness**
  - Interactive application
- **Resource sharing**
  - Address space and other resources
- **Economy: less overhead**
  - Solaris: process creation 30X overhead than thread;
  - Context switching threads within a process 5X faster
- **Scalability**
  - Better utilization of multiprocessor/multicore systems

Concurrency vs. Parallelism?

Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



Threads vs. Processes

- Threads and processes: **similarities**
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched (scheduled) by the kernel
- Threads and processes: **differences**
  - Threads share code and data, processes (typically) do not
  - Threads are less expensive than processes
    - Process control (creation and exit) is more expensive than thread control
    - Context switches for processes are more

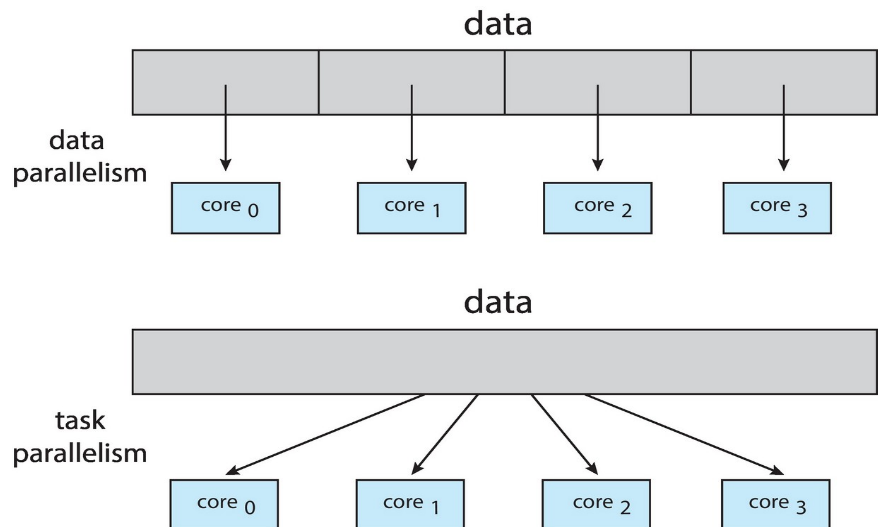
Pros and Cons of Thread-Based Designs

- + Easy to share data between threads
  - e.g., logging information, file cache
- + Threads are more efficient than processes
- – Unintentional sharing can introduce subtle and hard-to-

OPT

Multicore Programming

- Multithreaded programming provides a mechanism for efficient use of multicore systems
- Multicore programming will require entirely new approach to SW
- Programmers face new challenges
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**



- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each **[do(1-5) do(6-10) … do(90-100)]**
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation **[doA(1-100) doB(1-100) … doZ(1-100)]**

.

An Example: *testthread.c*

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS        5
void *PrintHello(void *threadid);
int main(int argc, char *argv[]){
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
         }
    } // wait for other threads
    for(t=0;t<NUM_THREADS;t++)  pthread_join(threads[t], NULL);
}
void *PrintHello(void *threadid){
   int i;
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   for(i=0; i < 10; i++) printf("It's thread #%ld and i=%d\n", tid, i);
   pthread_exit(NULL);       {int c=1000; while(c--); printf … }
}
```

To compile, link with the pthread library.

> gcc testthread.c -o testthread -lpthread

Give the possible outputs….

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
It's thread #0 and i=0
It's thread #0 and i=1
Hello World! It's me, thread #1!
It's thread #1 and i=0
It's thread #0 and i=2
It's thread #0 and i=3
It's thread #0 and i=4
It's thread #1 and i=1
It's thread #1 and i=2
It's thread #1 and i=3
In main: creating thread 2
It's thread #0 and i=5
Hello World! It's me, thread #2!
It's thread #2 and i=0
It's thread #0 and i=6
It's thread #0 and i=7
```

 ….

Exercises on *testthread.c*

- **What if you make i in the function global?**
- How would you pass an array of doubles to this thread? Thread will add all and print the sum
- How about passing an integer, a double, and a character as parameters to this thread? Thread will simply print them.
- What if we wanted to get some results back from that thread, what would you do?

How to implement threads?

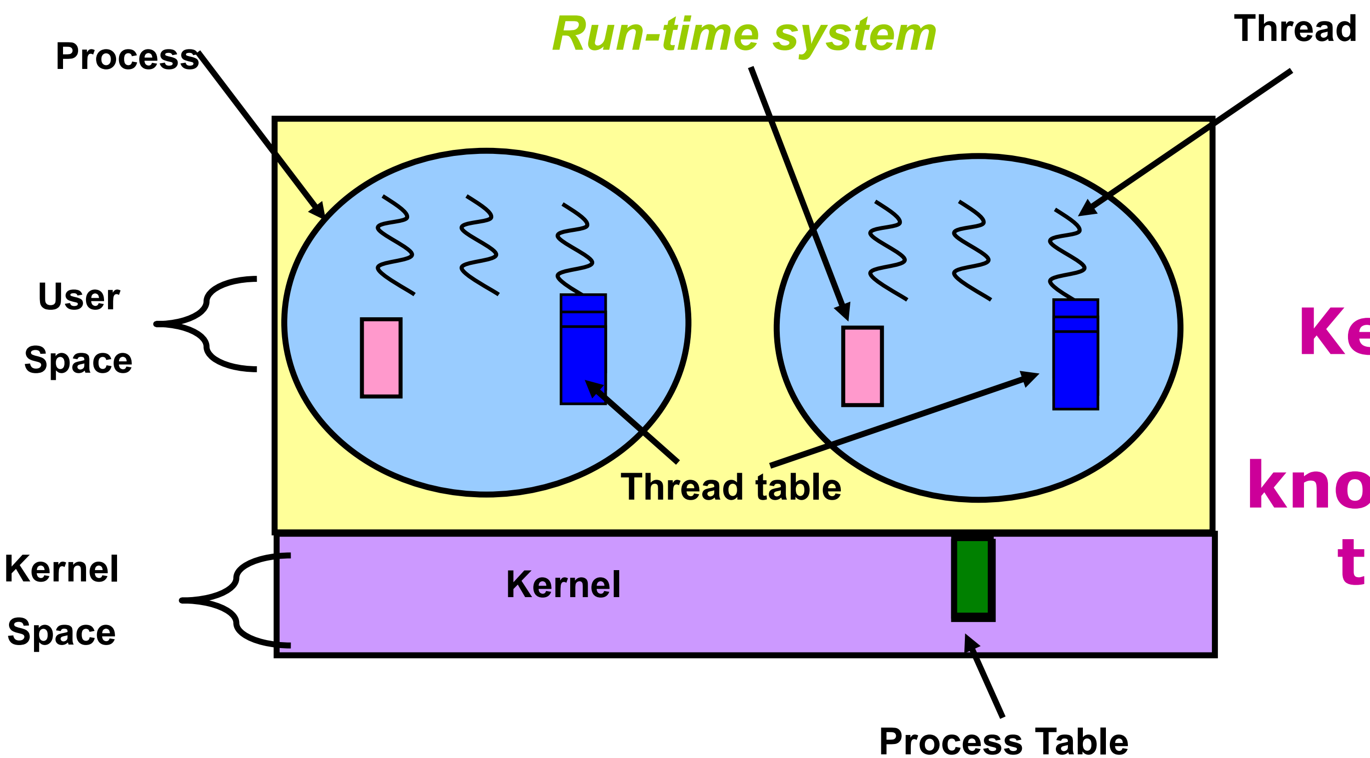
MULTI-THREADING MODELS

Thread Implementations: Issues

- Process usually starts with a single thread and creates others
- Thread management operations (similar to process management)
  - **Creation**: procedure/method for the new thread to run
  - **Scheduling**: runtime properties/attributes
  - **Destruction**: release resources
  - Thread Synchronization
    - join, wait, etc.
- ***Who*** manages threads and ***where***
  - **User space**: managed by applications using some libraries
  - **Kernel space**: managed by OS
    - all modern OSes have kernel level support (more effective, parallel exec)

User-Level Threads

- User threads: ***thread library*** at the <u>user level</u>
- **Run-time system** provides support for thread creation, scheduling and management



Kernel has *NO* knowledge of threads!

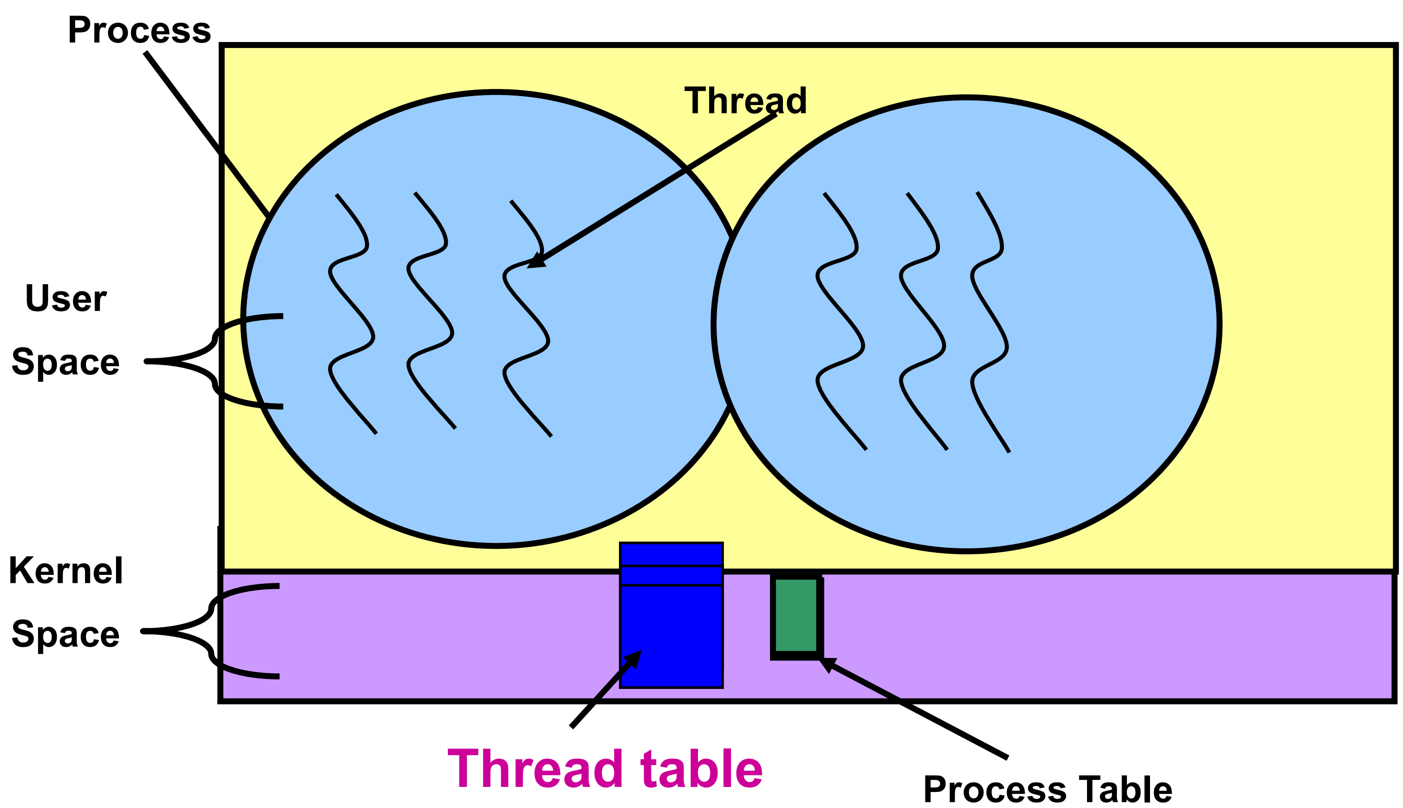
User-Level Threads (cont.)

- Each process needs its own **private *thread table*** to keep track of the threads in that process.
- The thread-table keeps track of the per-thread items (program counter, stack pointer, register, state..)
- When a thread does something that *may* cause it to become blocked ***locally*** (e.g. wait for another thread), it calls a **run-time system** procedure.
- If the thread must be put into blocked state, the procedure performs ***thread switching***
- Advantages
  - **Fast** thread switching: no kernel activity involved
  - Customized/**flexible** thread scheduling algorithm
  - Application **portability**: different machines with library
- Problems/disadvantages:
  - Kernel only knows process
  - **Blocking** system calls: kernel blocks process, so one thread blocks all activities (many-to-one mapping)
  - All threads share **one CPU,** so cannot use multi-proc./core

Kernel-Level Threads

Supported directly by OS

Kernel performs thread creation, scheduling & management in kernel space



Kernel-Level Threads (cont.)

- Advantages
  - User activity/thread with blocking I/O does NOT block other activities/threads from the same user
  - When a thread blocks, the kernel may choose another thread from the **same** or **different** process
  - Multi-activities in applications can use multi-cores
- Problems/disadvantages
  - Thread management could be relatively costly:
    - all methods that might block a thread are implemented as system calls
  - Non-flexible scheduling policy
  - Non-portability: application can only run on same type of machine

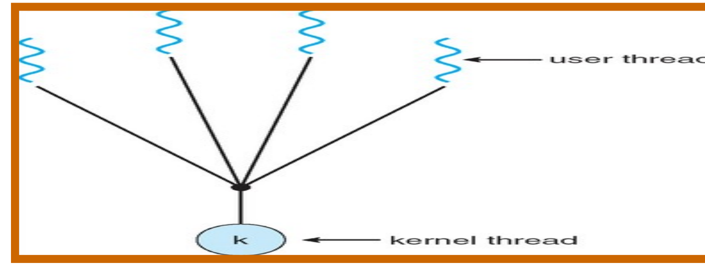
What is the relationship between **user** level and **kernel** level threads?

How to map **user** level threads to **kernel** level threads?

Mapping: User ☾ Kernel Threads

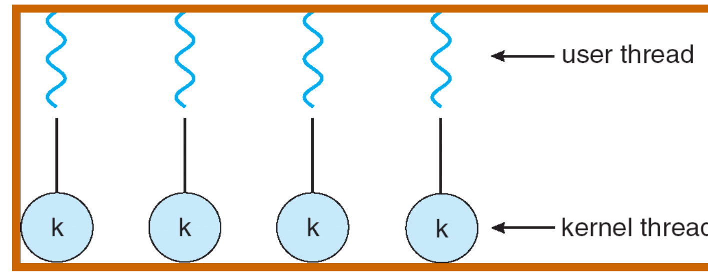
- **Many-to-one**
  - Many user threads ☾ one kernel thread (-/+ are same as in user-level threads)

    Examples: Solaris Green Threads, GNU Portable Threads



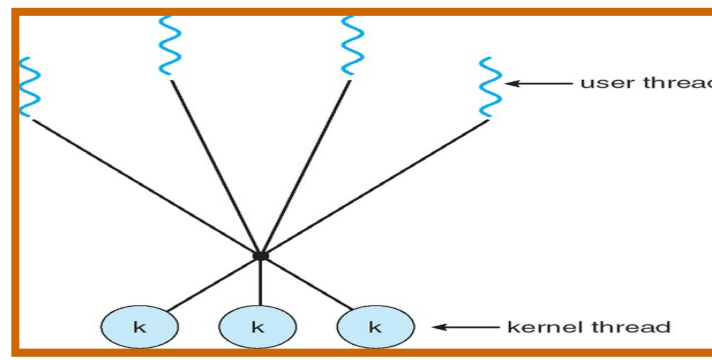
- **One-to-One**
  - One user thread ☾ one kernel thread;
  - + more concurrency
  - - limited number of kernel threads

  Examples: Windows NT/XP/2000, Linux, Solaris 9 and later



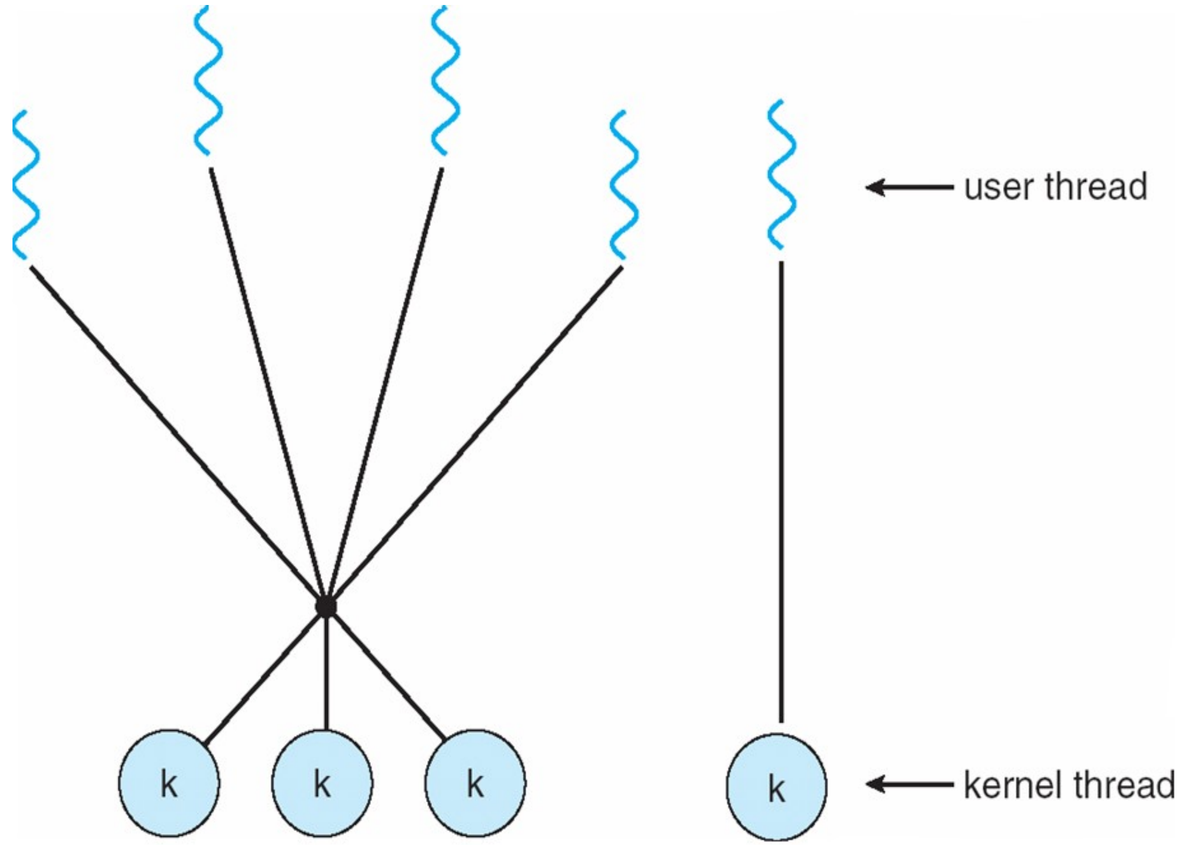
- **Many-to-Many**
  - Many user threads ☾ many kernel threads
  - + no limit on the number of user threads
  - - not true concurrency because kernel has limited number of threads

    Examples: Solaris prior to version 9, Windows NT/2000 with the ThreadFiber package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

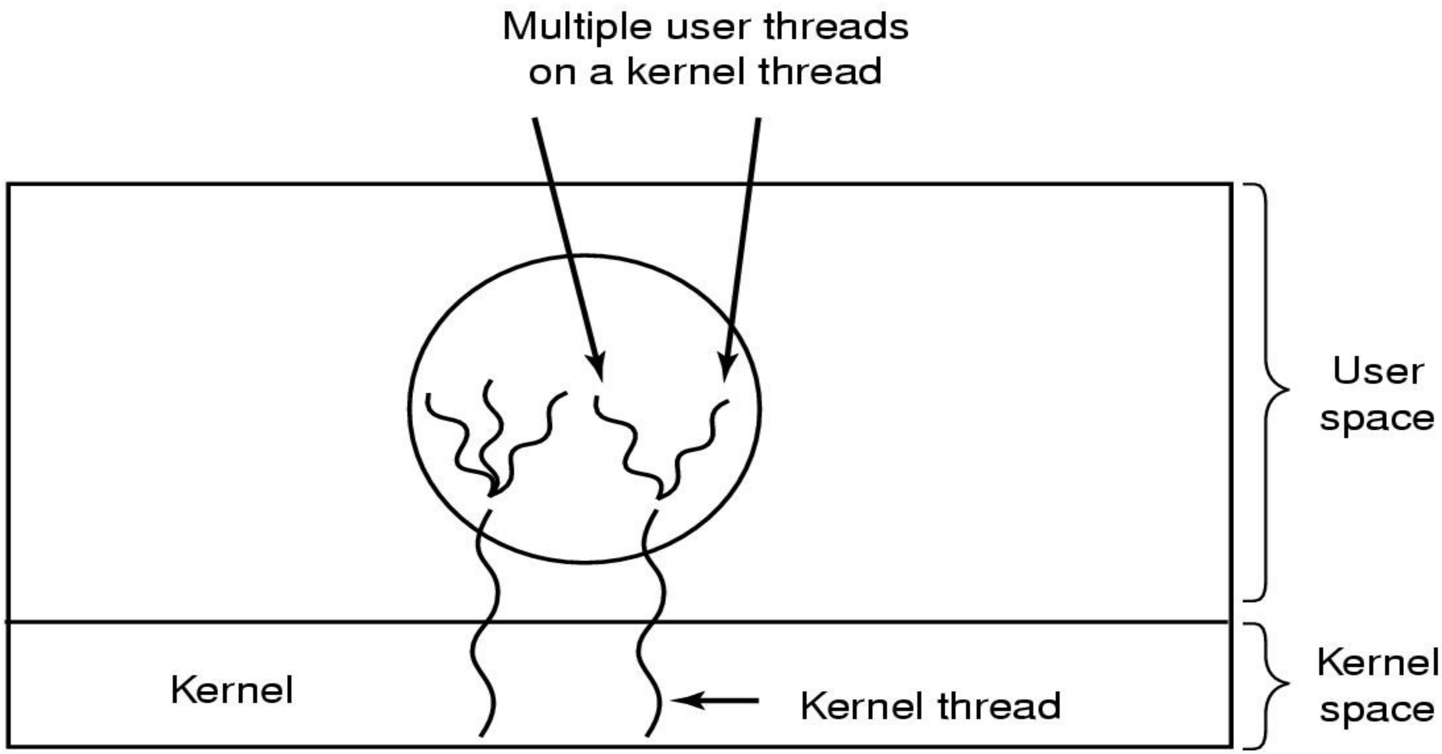


*

Hybrid Implementation

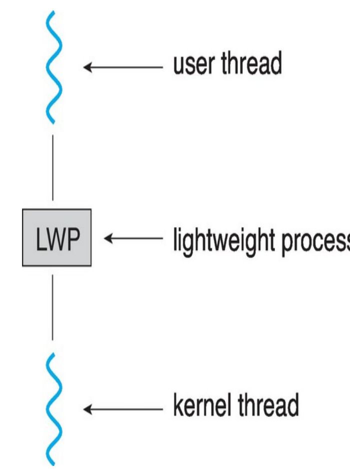
combine the best of both approaches

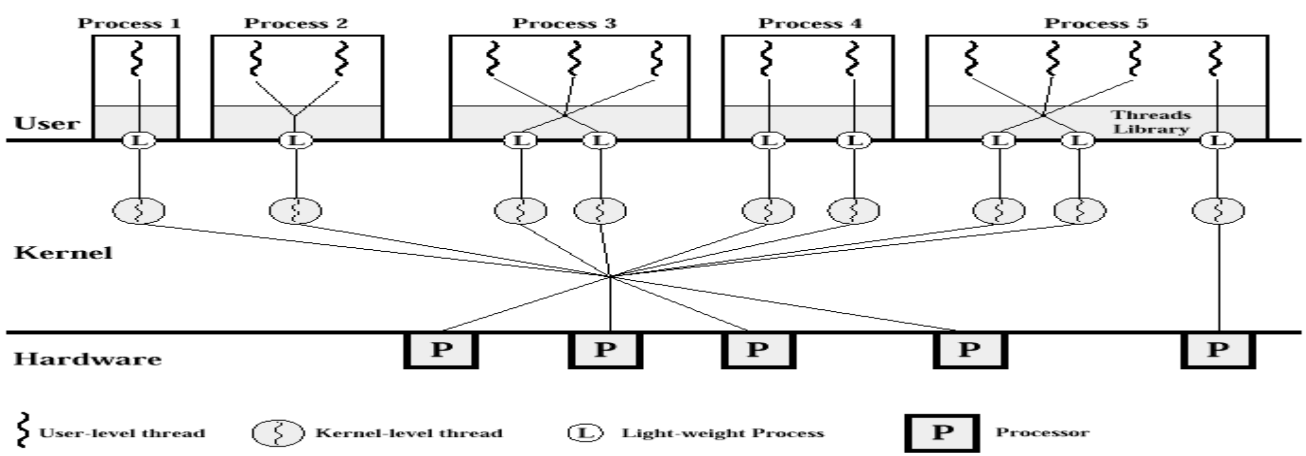
- Use kernel-level threads, and then multiplex user-level threads onto some or all of the kernel threads.



Light-Weight Process (LWP)

- **Lightweight process (LWP):** intermediate data structure
  - For user-level threads, LWP is a **Virtual processor**
  - **Each LWP attaches to a kernel thread**
- Multiple user-level threads ☾ a single LWP
  - Normally from the same process
- **A process may be assigned multiple LWPs**
  - Typically, an LWP for each blocking system call
- OS schedules kernel threads (hence, LWPs) on the CPU





LWP: Advantages and Disadvantages

- + User level threads are easy to create/destroy/sync
- + A blocking call will not suspend the process if we have enough LWP
- + Application does not need to know about LWP
- +LWP can be executed on different CPUs, hiding multiprocessing

- - Occasionally, we still need to create/destroy LWP (as expensive as kernel threads)
- - Makes **upcalls** (scheduler activation)
  - + simplifies LWP management
  - - Violates the layered structure

Provide programmers with API for creating and managing threads

THREAD LIBRARIES

USP Chapter 12

Thread Libraries

- Thread Libraries can be implemented in two ways:
  - **User-level** library
    - Entirely in user space
    - Everything is done using thread related function calls (no syscalls)
  - **Kernel-level** library supported by the OS
    - Code and data structures for kernels are in kernel space
    - Thread related function calls result in system calls to kernel
- Three primary thread libraries:
  - POSIX Threads **Pthread** *(either a user-level or kernel-level)*
  - Win32 threads *(kernel-level)*
  - Java threads *(JVM manages threads by using host system threads)*
    - *Threads are fundamental model of prog exec,*
    - *Java provides rich set of features for thread creation and mng.*

POSIX Threads: Pthread

- POSIX
  - **P**ortable **O**perating **S**ystem **I**nterface [for Uni**x**]
  - Standardized programming interface
- **Pthread**
  - 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 (Defined as a set of C types and procedure calls)
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- Implementations
  - User-level vs. kernel-level
- More information about Pthread programming

  https://computing.llnl.gov/tutorials/pthreads/

Pthread APIs: Four Groups

U S P C h 1 3 - S G G C h 5

- **Thread management**
  - Routines for creating, detaching, joining, etc.
  - Routines for setting/querying thread attributes
- **Mutexes**: abbreviation for "mutual exclusion"
  - Routines for creating, destroying, locking/unlocking
  - Functions to set or modify attributes with mutexes.
- **Conditional variables**
  - Communications for threads that share a mutex
  - Functions to create, destroy, wait and signal based on specified variable values
  - Functions to set/query condition variable attributes
- **Synchronization**
  - Routines that manage read/write locks and barriers

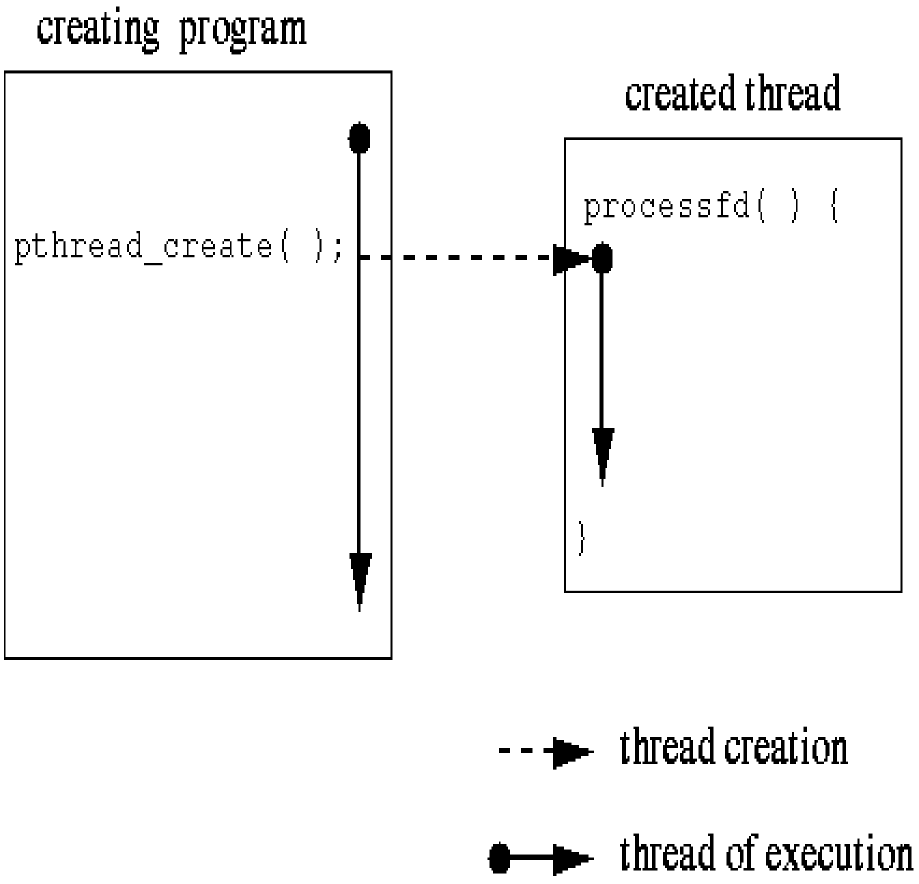
Thread Call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Thread Creation

#include <pthread.h>

pthread_t threadID;

*pthread_create (&threadID, NULL, methodName, *para);*



- 1st argument is the ID of the new thread
- 2nd argument is a pointer to pthread_attr_t
- 3rd argument is **thread (function/method) name**
- 4th argument is a pointer to the arguments for the thread's method/function
- When successful, returns 0

An Example: *testthread.c*

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid){
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[]){
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   for(t=0;t<NUM_THREADS;t++)
     pthread_join(threads[t], NULL);  // wait for other threads
}
```

To compile, link with the pthread library.

> gcc testthread.c -o test –lpthread

```
> test
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

Associated Function: Work for the Thread

- Need to have a special format

```
void *ThFunc(void *arg);
```

- Take a single parameter (void *)
  - Allow any pointer to be passed
  - Point to a structure (with any number of parameters)
- Return void *

```
void *ThFunc(void *arg){
    long tid;

    tid = (long) arg;

    printf("My ID %ld!\n", tid);

    // do something

    pthread_exit(NULL);
}
```

```
int error;  int fd;  pthread_t tid;
if (error=pthread_create(&tid, NULL, ThFunc, &fd))
   fprintf(stderr, "Failed to create thread: %s\n",
                                      strerror(error));
```

When successful, returns 0

Process Exit and Thread Exit

- A **process** terminates when
  - it calls exit directly
  - one of its threads calls exit
  - it executes return from main
  - it receives a signal
- When a process terminates, all its threads terminate
- Thread exit/cancellation
  - Call `return` from its main thread function or `pthread_exit`

```
void pthread_exit(void *value_ptr);
```

  - Can request another thread exit with `pthread_cancel`

```
int pthread_cancel(pthread_t thread);
```

More about thread cancellation later…

Thread: Join

- A thread normally exits when its function returns
- Some resources of a thread stay around when it exists until it has been joined (waited for) or its process terminates
- A thread waits for (kid) threads with `pthread_join`

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

Why do we need double pointer here?

Return Value of A Thread

- The pthread_join function suspends the calling function until a specific thread has completed
- A thread can pass a value to another thread in a process through its return value
- The return value is a ***pointer to arbitrary*** data
  - Threads in a process share an address space

```
int error;
int *exitcodep;
pthread_t tid;

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 ID

- Each thread has an ID of type `pthread_t`
  - It is just an integer (like a process ID) on most systems
- Retrieve a thread's own ID with `pthread_self`
- Thread IDs can be compared with `pthread_equal`

```
pthread_t pthread_self(void);

int pthread_equal(thread_t t1, pthread_t t2);
```

Thread: Detach

- By default a thread is attached to the parent thread
  - Parent should wait (join) to release child thread’s resources

- But it is possible to detach a thread
  - make it release its own resources when it terminates
  - A detached thread cannot be joined.

- Detach a thread with `pthread_detach`

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

Examples: Detach A Thread

Example 1: Create and detach a thread

```
void *processfd(void *arg);

int error;
int fd
pthread_t tid;

if (error = pthread_create(&tid, NULL, processfd, &fd))
    fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
else if (error = pthread_detach(tid))
    fprintf(stderr, "Failed to detach thread: %s\n", strerror(error));
```

Example 2: A thread detach itself

```
void *detachfun(void *arg) {
   int i = *((int *)(arg));
   if (!pthread_detach(pthread_self()))
      return NULL;
   fprintf(stderr, "My argument is %d\n", i);
   return NULL;
}
```

PARAMETER PASSING TO THREADS

*****

Parameters and Return Values of Threads

- Parameter: a void pointer
  - to an array of <u>same-type</u> parameters,
  - to a structure of <u>different-type</u> parameters,
  - type cast <u>long</u> as pointers
- Return value: a void pointer
  - Terminating thread passes a pointer to the joining thread
  - The threads share the same address space.
  - The return value must exist after the thread terminates; **Must not use the address of automatic variable** in thread for return value

```
void *ThFunc(void *arg){
   // get some data

   // do some work…

   // return  some data
}
```

```
long t=200, ret;
long *ptrret;
pthread_create(&tid, NULL,
               ThFunc,(void *)t);
t=300;
…
pthread_join(tid, &ptrret);
ret = *ptrret
```

```
void *ThFunc(void *arg){
    long myt;
    myt = (long) arg;
    printf("ID %ld!", myt);
    pthread_exit(NULL);
}
```

*****

Parameters of Threads (long) – special case

```
long t=200;

pthread_create(&tid, NULL,
                ThFunc,(void *)t);
t=300;
…
```

```
void *ThFunc(void *arg){
    long myt;
    myt = (long) arg;
    printf("ID %ld!", myt);
    pthread_exit(NULL);
}
```

*****

Parameters of Threads (double) – general case

```
double t=200.5;

pthread_create(&tid, NULL,
               ThFunc,(void *) &t);
// t=300.5;
…
```

```
void *ThFunc(void *arg){
    double myt;
    myt = *((double *)arg);
    printf("I got %lf!", myt);
    pthread_exit(NULL);
}
```

```
double t=200.5, *ptrt;

ptrt = (double *) malloc(sizeof(double));
if (ptrt == NULL) exit();
*ptrt = t;
pthread_create(&tid, NULL,
                ThFunc,(void *) ptrt);
t=300.5;
……
```

*****
exercise

Parameters of Threads (long) – general case

```
long t=200;

pthread_create(&tid, NULL,
                ThFunc,(void *)&t);
// t=300;
…
```

```
void *ThFunc(void *arg){
    long myt;
    myt = *((long *) arg);
    printf("ID %ld!", myt);
    pthread_exit(NULL);
}
```

```
long t=200, *ptrt;

ptrt = (long *) malloc(sizeof(long));
if (ptrt == NULL) exit();
*ptrt = t;
pthread_create(&tid, NULL,
                ThFunc,(void *) ptrt);
t=300;
……
```

RETURN A VALUE FROM A THREAD

Return a long value - special case

```
long t=200, ret;
long *ptrret;
pthread_create(&tid, NULL,
               ThFunc,(void *)t);
t=300;
…
pthread_join(tid, (void **)&ptrret);
ret = (long) ptrret;
```

```
void *ThFunc(void *arg){
  long myt, myret;
  myt = (long) arg;
  printf("ID %ld!", myt);
  myret = ……;
  pthread_exit((void *)myret);
}
```

*****

Return a double value – general case

```
double t=200.5, ret;
double *ptrret;
pthread_create(&tid, NULL,
               ThFunc,(void *) &t);
// t=300.5;
…
pthread_join(tid, (void **)&ptrret);
ret = *ptrret;
```

```
void *ThFunc(void *arg){
 double myt, myret;
 myt = *((double *)arg);
 printf("I got %lf!", myt);
 myret = ……;
 pthread_exit((void *)&myret);
}              // NO! WHY?
```

```
void *ThFunc(void *arg){
 double myt, myret, *myptrret;
 myt = *((double *)arg);
 printf("I got %lf!", myt);
 myret = ……;

 myptrret = (double *)
       malloc(sizeof(double));
 if(myptrret == NULL) exit();

 *myptrret = myret;
 pthread_exit((void *)myptrret);
}
```

Return a long value – general case

***** exercise

```
long t=200, ret;
long *ptrret;
pthread_create(&tid, NULL,
                 ThFunc,(void *)t);
t=300;
…
pthread_join(tid, (void **)&ptrret);
ret = *ptrret;
```

```
void *ThFunc(void *arg){
 long myt, myret;
 myt = (long) arg;
 printf("ID %ld!", myrt);
 myret = ……;
 pthread_exit((void *)&myret);
}              // NO! WHY?
```

```
void *ThFunc(void *arg){
 long myt, myret, *myptrret;
 myt = (long)arg;
 printf("I got %lf!", myt);
 myret = ……;

 myptrret = (long *)
          malloc(sizeof(long));
 if (myptrret==NULL)exit();
 *myptrret = myret;
 pthread_exit((void *)myptrret);
}
```

Exercises on passing parameters and returning values

- DO psq04 (parameter pass/return long, double),

*****

How to Pass STRUCT to Threads

```
typedef struct param {
  int x; double y; char z;
} param_t;

param_t t={4, 5.6, 'a'};

pthread_create(&tid, NULL,
               ThFunc,(void *) &t);
…
```

```
typedef struct param {
  int x; double y; char z;
} param_t;

param_t t={4, 5.6, 'a'}, *ptrt;

ptrt = malloc(…); // if NULL
*ptrt = t;
pthread_create(&tid, NULL,
        ThFunc,(void *) ptrt);
t.x=2;
…
```

```
void *ThFunc(void *arg){
 param_t myt, *myptrt;

 myt = *((param_t *)arg);
 myptrt = (param_t *)arg;

 printf("Got %lf!", myt.y);
 printf("Got %lf!", myptrt->y);

}
```

*****

How to Return STRUCT from Threads

```
typedef struct param {
   int x; double y; char z;
} param_t;

param_t t={4, 5.6, 'a'}, ret;
param_t *ptrret;
pthread_create(&tid, NULL,
               ThFunc,(void *) &t);
…
pthread_join(tid, (void **)&ptrret);
ret = *ptrret;
```

```
void *ThFunc(void *arg){
 param_t myt, *myptrt;
 param_t myret, *myptrret;
 myt = *((param_t *)arg);
 myptrt = (param_t *)arg;

 printf("Got %lf!", myt.y);
 printf("Got %lf!", myptrt->y);

 myret.x = ……;

 myptrret = (param_t *)
        malloc(sizeof(param_t));
 if(myptrret == NULL) exit();

 *myptrret = myret;
 pthread_exit((void *)myptrret);
}
```

*****

How to Pass an ARRAY to Threads

```
a_type  arr[100];

pthread_create(&tid, NULL,
               ThFunc,(void *) arr);
…
```

```
void *ThFunc(void *arg){
 a_type *myarr;
 myarr = (a_type *)arg;

 myarr[i] ......

}
```

*****

How to Return an ARRAY from Threads

```
a_type  arr[100], *ret;
a_type **ptrret;
pthread_create(&tid, NULL,
               ThFunc,(void *) arr);
…
pthread_join(tid, (void **)&ret);

ret[i] ........
```

```
void *ThFunc(void *arg){
 a_type *myarr, myret[5], *myptrret;
 myarr = (a_type *)arg;

 myarr[i] ......

 myret[i] = ……;

 myptrret = (a_type *)
       malloc( 5 * sizeof(a_type));
 if(myptrret == NULL) exit();
 myptrret[i] = myret[i];  ………

 pthread_exit((void *)myptrret);
}
```

Exercise: An example to pass an array of int and get return an int value from a thread!

```
void *copyfilemalloc(void *arg);

int main (int argc, char *argv[]) {          /* copy fromfile to tofile */
   int *bytesptr;
   int error;
   int fds[2];
   pthread_t tid;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s fromfile tofile\n", argv[0]);
      return 1;
   }
   if (((fds[0] = open(argv[1], READ_FLAGS)) == -1) ||
       ((fds[1] = open(argv[2], WRITE_FLAGS, PERMS)) == -1)) {
      perror("Failed to open the files");
      return 1;
   }
   if (error = pthread_create(&tid, NULL, copyfilemalloc, fds)) {
      fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
      return 1;
   }
   if (error = pthread_join(tid, (void **)&bytesptr)) {
      fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
      return 1;
   }
   printf("Number of bytes copied: %d\n", *bytesptr);
   return 0;
}
```

What will happen if we don't use pthread_join(…)?

```c
void *copyfilemalloc(void *arg);

int main (int argc, char *argv[]) {          /* copy fromfile to tofile */
    int *bytesptr;
    int error;
    int fds[2];
    pthread_t tid;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s fromfile tofile\n", argv[0]);
        return 1;
    }
    if (((fds[0] = open(argv[1], READ_FLAGS)) == -1) ||
        ((fds[1] = open(argv[2], WRITE_FLAGS, PERMS)) == -1)) {
        perror("Failed to open the files");
        return 1;
    }
    if (error = pthread_create(&tid, NULL, copyfilemalloc, fds)) {
        fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
        return 1;
    }
    if (error = pthread_join(tid, (void **)&bytesptr)) {
        fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
        return 1;
    }
    printf("Number of bytes copied: %d\n", *bytesptr);
    return 0;
}

void *copyfilemalloc(void *arg)  { /* copy infd to outfd with return value */
    int *bytesp;
    int infd;
    int outfd;

    infd = *((int *)(arg));
    outfd = *((int *)(arg) + 1);
    if ((bytesp = (int *)malloc(sizeof(int))) == NULL)
        return NULL;
    *bytesp = copyfile(infd, outfd);
    r_close(infd);
    r_close(outfd);
    return bytesp;
}
```

```c
int *arr;
arr = (int *) arg;
infd  = arr[0];
outfd = arr[1];
```

An Alternate Approach

- Calling thread pre-allocate the space for return value

```
void *copyfilepass(void *arg)  {
   int *argint;

   argint = (int *)arg;
   argint[2] = copyfile(argint[0], argint[1]);
   r_close(argint[0]);
   r_close(argint[1]);
   return argint + 2;
}
```

```
void *copyfilepass(void *arg);

int main (int argc, char *argv[]) {
   int *bytesptr;
   int error;
   int targs[3];
   pthread_t tid;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s fromfile tofile\n", argv[0]);
      return 1;
   }

   if (((targs[0] = open(argv[1], READ_FLAGS)) == -1) ||
       ((targs[1] = open(argv[2], WRITE_FLAGS, PERMS)) == -1)) {
      perror("Failed to  open the files");
      return 1;
   }
   if (error = pthread_create(&tid, NULL, copyfilepass, targs)) {
      fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
      return 1;
   }
   if (error = pthread_join(tid, (void **)&bytesptr)) {
      fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
      return 1;
   }
   printf("Number of bytes copied: %d\n", *bytesptr);
   return 0;
}
```

Exercise: What can be the output?

Program 12.9: badparameters.c, page 429

Bad Parameters

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#define NUMTHREADS 10

static void *printarg(void *arg) {
   fprintf(stderr, "Thread received %d\n", *(int *)arg);
   return NULL;
}

int main (void) {          /* program incorrectly passes parameters to threads */
   int error;
   int i;
   int j;
   pthread_t tid[NUMTHREADS];

   for (i = 0; i < NUMTHREADS; i++)
      if (error = pthread_create(tid + i, NULL, printarg, (void *)&i)) {
         fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
         tid[i] = pthread_self();
      }
   for (j = 0; j < NUMTHREADS; j++) {
      if (pthread_equal(pthread_self(), tid[j]))
         continue;
      if (error = pthread_join(tid[j], NULL))
         fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
   }
   printf("All threads done\n");
   return 0;
}
```

Exercise: What can you return from this thread?

What can be put instead of **NULL** in the last line?

```
void *whichexit(void *arg) {
   long n;
   long np1[1];
   long *np2;
   char s1[10];
   char s2[] = "I am done";
   n = 3;
   np1[0] = n;
   np2 = (int *)malloc(sizeof(int));
   *np2 = n;
   strcpy(s1, "Done");
   return( NULL );
}
```

1. `n`
2. `&n`
3. **`(long *)n`**
4. `np1`
5. **`np2`**
6. `s1`
7. `s2`
8. **`"This works"`**
9. `strerror(EINTR)`

Suppose we have
`char *s2 = "I am done";`
10. `s2` vs. number 7

Solution to Exercise…

- 1. The return value is a pointer, not an integer, so this is invalid.
- 2. The integer n has automatic storage class, so it is illegal to access it after the function terminates.
- 3. This is a common way to return an integer from a thread. The integer is cast to a pointer. When another thread calls pthread_join for this thread, it casts the pointer back to an integer. While this will probably work in most implementations, **it should be avoided.** The C standard [56, Section 6.3.2.3] says that an integer may be converted to a pointer or a pointer to an integer, but the result is implementation defined. It does not guarantee that the result of converting an integer to a pointer and back again yields the original integer.
- 4. The array np1 has automatic storage class, so it is illegal to access the array after the function terminates.
- 5. This is safe since the dynamically allocated space will be available until it is freed.
- 6. The array s1 has automatic storage class, so it is illegal to access the array after the function terminates.
- 7. The array s2 has automatic storage class, so it is illegal to access the array after the function terminates.
- 8. This is valid in C, since string literals have static storage duration.
- 9. This is certainly invalid if strerror is not thread-safe. Even on a system where strerror is thread-safe, the string produced is not guaranteed to be available after the thread terminates.

Exercises on passing parameters and returning values

- See quiz exercises
  - psq05 (parameter pass/return struct, array)

A hidden problem with threads is that they may call library functions that are not thread-safe, possibly producing faulty results.

THREAD SAFETY

REF

Thread Safety: Allow Threads Re-Entry

Allow multiple threads to execute the functions at the same time;

Almost all system and library functions are safe for threads;

Except the ones in the right table:

asctime	fcvt	getpwnam	nl_langinfo
basename	ftw	getpwuid	ptsname
catgets	gcvt	getservbyname	putc_unlocked
crypt	getc_unlocked	getservbyport	putchar_unlocked
ctime	getchar_unlocked	getservent	putenv
dbm_clearerr	getdate	getutxent	pututxline
dbm_close	getenv	getutxid	rand
dbm_delete	getgrent	getutxline	readdir
dbm_error	getgrgid	gmtime	setenv
dbm_fetch	getgrnam	hcreate	setgrent
dbm_firstkey	gethostbyaddr	hdestroy	setkey
dbm_nextkey	gethostbyname	hsearch	setpwent
dbm_open	gethostent	inet_ntoa	setutxent
dbm_store	getlogin	l64a	strerror
dirname	getnetbyaddr	lgamma	strtok
dlerror	getnetbyname	lgammaf	ttyname
drand48	getnetent	lgammal	unsetenv
ecvt	getopt	localeconv	wcstombs
encrypt	getprotobyname	localtime	wctomb
endgrent	getprotobynumber	lrand48	
endpwent	getprotoent	mrand48	
endutxent	getpwent	nftw	

Example (from the first few weeks)

`int count=0;`
`char *next_label()`
`{`
`    static char b[10];`
 	
`    count++;`
`    sprintf(b,"Fig. %d",`
`                count);`
`    return b;`
`}`
	`char *next_label()`
	`{`
	`    static int count=0;`
	`    char *b;`
	`    b = malloc(10);`
	`    if (!b) exit();`
	`    count++;`
	`    sprintf(b,"Fig. %d",`
	`                count);`
	`    return b;`
	`}`

How about count++? Is there anything wrong with that statement?

THREAD ATTRIBUTES

Thread Attributes

REF

pthread_t threadID;

pthread_attr_t attr;

…

*pthread_create (&threadID, NULL, methodName, *para);*

------------------------------ &attr ---------------------------

- Attributes of a thread behave like objects, which can be created or destroyed
  - Create an attribute object (initialize with default properties)
  - Modify the properties of the attribute object.
  - Create a thread using the attribute object.
  - It can be changed/reused without affecting thread.
  - It affects the thread only at time of thread creation.

```
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_init(pthread_attr_t *attr);
```

Thread Attributes (cont.)

REF

- Settable properties of thread attributes

property	function
attribute objects	pthread_attr_destroy
	pthread_attr_init
state	pthread_attr_getdetachstate
	pthread_attr_setdetachstate
stack	pthread_attr_getguardsize
	pthread_attr_setguardsize
	pthread_attr_getstack
	pthread_attr_setstack
scheduling	pthread_attr_getinheritsched
	pthread_attr_setinheritsched
	pthread_attr_getschedparam
	pthread_attr_setschedparam
	pthread_attr_getschedpolicy
	pthread_attr_setschedpolicy
	pthread_attr_getscope
	pthread_attr_setscope

Thread States

REF

- States of a thread
  - `PTHREAD_CREATE_JOINABLE` (the default)
  - `PTHREAD_CREATE_DETACHED`
- Get/set the state of a thread

```
int pthread_attr_getdetachstate(const pthread_attr_t
                                *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
```

```
int error, fd;
pthread_attr_t tattr;
pthread_t tid;

if (error = pthread_attr_init(&tattr))
   fprintf(stderr, "Failed to create attribute object: %s\n",
                    strerror(error));
else if (error = pthread_attr_setdetachstate(&tattr,
                 PTHREAD_CREATE_DETACHED))
   fprintf(stderr, "Failed to set attribute state to detached: %s\n",
           strerror(error));
else if (error = pthread_create(&tid, &tattr, processfd, &fd))
   fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
```

Stack of A Thread

- Set the location and size for the thread stack

```
int pthread_attr_getstack(const pthread_attr_t *restrict
                  attr, void **restrict stackaddr,
                  size_t *restrict stacksize);
int pthread_attr_setstack(pthread_attr_t *attr, void
                  *stackaddr, size_t stacksize);
```

- Set a guard for stack to protect overflow

```
int pthread_attr_getguardsize(const pthread_attr_t
            *restrict attr, size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
                        size_t guardsize);
```

Contention Scope of A Thread **

- The `contentionscope` of a thread
  - `PTHREAD_SCOPE_PROCESS`
  - `PTHREAD_SCOPE_SYSTEM`
  - Determines whether thread competes with other threads within a process or with other processes in the system

- Get/set `contentionscope` of a thread

```
int pthread_attr_getscope(const pthread_attr_t *restrict
                          attr, int *restrict contentionscope);
int pthread_attr_setscope(pthread_attr_t *attr, int
                          contentionscope);
```

Thread Scheduling

REF

- Inheritance of scheduling policy:
  - `PTHREAD_INHERIT_SCHED`: the attribute object's scheduling attributes are ignored and the created thread has the **same scheduling attributes** of main thread.
  - `PTHREAD_EXPLICIT_SCHED:` the scheduling is taken from the attribute.
- Get/set Inheritance of scheduling policy

```
int pthread_attr_getinheritsched(const pthread_attr_t
      *restrict attr, int *restrict inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                      int inheritsched);
```

Thread Scheduling (cont.)

REF

- Typical **scheduling policies**:
  - `SCHED_FIFO`
  - `SCHED_RR` and
  - `SCHED_OTHER`
  - Support is system dependent
- **Scheduling parameters** for each policy are stored in `struct sched_param`: priority or quantum value

```
int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
                               struct sched_param *restrict param);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                               const struct sched_param *restrict param);
int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
                                    int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

THIS PART (slides 68-79) IS FOR SELF-STUDY

Needed for next assignment

Threads are fundamental model of program execution in Java. So, Java provides a rich set of features for thread creation and management

JAVA THREADS

http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

How to Create Threads in Java (1)

There are two ways in Java:

1. Create a class `MyTh` that directly extends `Thread` class
   - The code in `MyTh.run()` will be the new thread
   - Then in a driver program
     - `MyTh th = new MyTh( … );`
     - `th.start(); // not th.run();`
   - Not recommended (why?)
     - A bad habit for industrial strength development
     - The methods of the worker class and the Thread class get all tangled up
     - Makes it hard to migrate to Thread Pools and other more efficient approaches

```
public class MyTh
              extends Thread {
  public MyTh(…){
  …
  }
  public void run() {
   //overwrite this …
  }
}
```

```
public class Thread {
…
public String getName();
public void interrupt();
public boolean isAlive();
public void join();
public void setDaemon(boolean on);
public void setName(String name);
public void setPriority(int level);
public static Thread currentThread();
public static void sleep(long ms);
public static void yield();
}
```

How to Create Threads in Java (2)

2. Define a class `MyTh` that implements `Runnable` interface
   - The code in `MyTh.run()` will be the new thread
   - Then in a driver program
     - `Thread th = new Tread( new MyTh(…) );`
     - `th.start();  // not th.run(); why? What happens?`

```
public interface Runnable
{
    public abstract void run();
}
```

```
public class MyTh
         implements Runnable {
   public MyTh(…){
      …
   }
   public void run() {
    //overwrite this …
   }
}
```

Example 1: Extend Thread Class

```
public class SimpleThread extends Thread {
  String msg;
  int repetition;
  public SimpleTread(String msg, int r){
      this.msg = msg;
      this.repetition = r;
  }
  public void run() {
     //overwrite run method
      for (int i = 0; i < repetition; i++)
         System.out.println("[" + i + "]" + msg);
  }
}
```

```
[0]T1
[0]T2
[1]T1
[1]T2
[2]T1
[2]T2
[3]T1
[3]T2
[4]T1
…
```

```
public class SimpleThreadMain {
  public static void main(String[] args) {
      SimpleThread t1 = new SimpleThread("T1", 100);
      t1.start();

      SimpleThread t2 =  new SimpleThread("T2", 100);
      t2.start();
  }
}
```

Example 2: Implement Runnable interface

```
public class SimpleRunnable implements Runnable {
   String msg;
   int repetition;
   public SimpleRunnable(String msg, int r) {
      this.msg = msg;
      this.repetition = r;
   }
   public void run( ) {
      //overwrite run method
      for (int i = 0; i < repetition; i++)
         System.out.println("[" + i + "]" + msg);
   }
}
```

```
[0]T1
[0]T2
[1]T1
[1]T2
[2]T1
[2]T2
[3]T1
[3]T2
[4]T1
…
```

```
public class SimpleRunnableMain {
  public static void main(String[] args) {
      SimpleRunnable r1 = new SimpleRunnable("T1", 100);
      Thread t1 = new Thread(r1);
      t1.start( );

      SimpleRunnable r2 = new SimpleRunnable("T2", 100);
      Thread t2 = new Thread(r2);
      t2.start( );
   }
}
```

Use *join()* to wait for a thread to finish

http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

The `join` method of `Thread` throws `InterruptedException` and must be placed in a `try-catch`.

```
…
  try {
      t1.join( );
      t1.join( );
   } catch (InterruptedException e) {}

   System.out.println("Both are done ");
```

```
for (int i = 0; i < 100; i++)
   System.out.println("main");
```

main
[0]T1
[0]T2
[1]T1
main
[1]T2
[2]T1
[2]T2
[3]T1
main
[3]T2
[4]T1
……
Both are done

[0]T1
[0]T2
[1]T1
[1]T2
[2]T1
[2]T2
[3]T1
[3]T2
[4]T1
……
main
main
main
……
Both are done

Java Thread Example - Output

```
public class ThreadExample implements Runnable {
   public void run() {
         for (int i = 0; i < 3; i++)
               System.out.println(i);
   }
   public static void main(String[] args) {
         new Thread( new ThreadExample()).start();
         new Thread( new ThreadExample()).start();
         System.out.println("Done");
   }
}
```

What are the possible outputs?

0,1,2,0,1,2,Done // thread 1, thread 2, main()

0,1,2,Done,0,1,2 // thread 1, main(), thread 2

Done,0,1,2,0,1,2 // main(), thread 1, thread 2

0,0,1,1,2,Done,2 // main() & threads interleaved

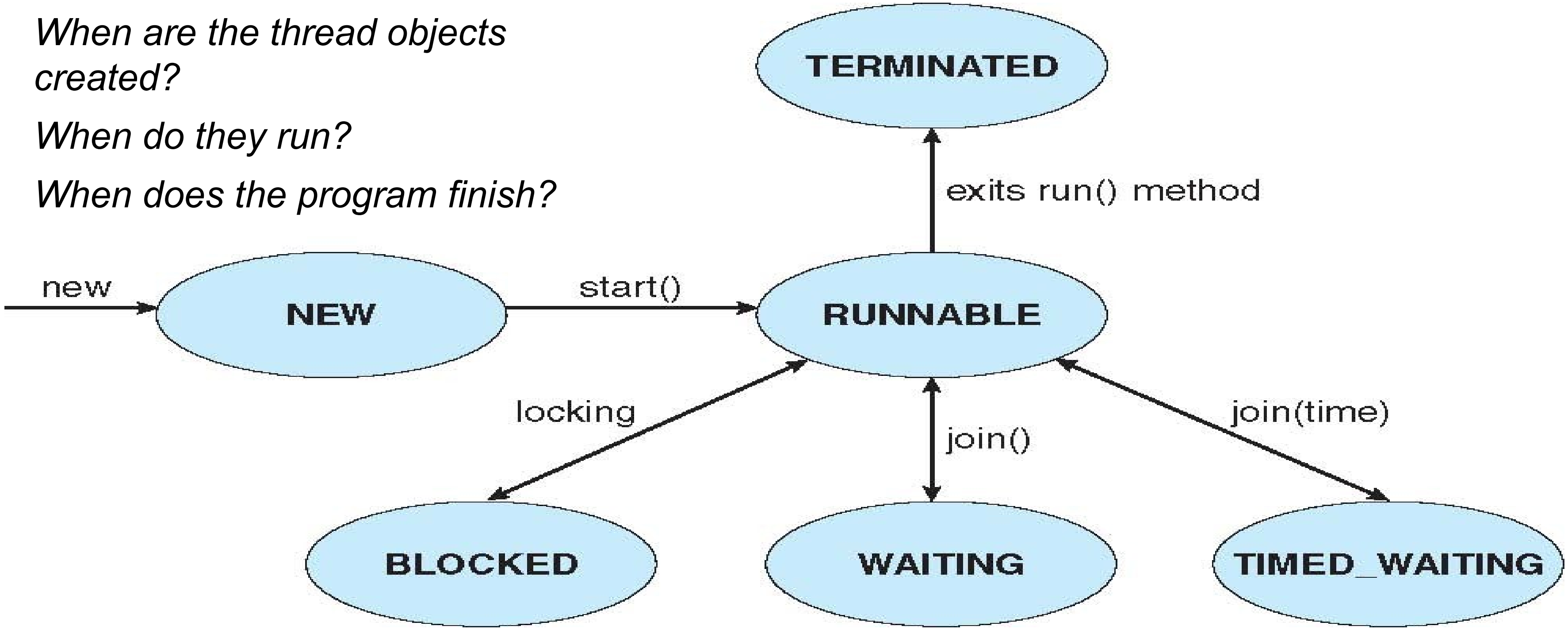
Why doesn't the program quit as soon as "Done" is printed?

JVM shuts down when all non-daemon threads terminate!

In pthread, when main quits all threads are terminated!

Life-Time of Java Threads

- *When are the thread objects created?*
- *When do they run?*
- *When does the program finish?*



- *A thread object exists when it is constructed, but it doesn't start running until the* start *method is called.*
- *A thread completes (or dies) when its run method finishes or when it throws an exception. The object representing this thread can still be accessed.*

Transitions between Threads

- Transitions between states caused by
  - Invoking methods in class Thread
    - start(), yield(), sleep(), wait(), join()
    - The `join`, `wait`, and `sleep` methods of `Thread` throw `InterruptedException` and must be placed in a `try-catch`.
- Other (external) events
  - Scheduler, I/O, returning from run()…
- Scheduler (ch5)
  - Determines which runnable threads to run
  - Part of OS or Java Virtual Machine (JVM)
  - Many computers can run multiple threads simultaneously (or nearly so)

Another Example: Java Threads

Define a class that implements **Runnable** interface

```
class MutableInteger
{
  private int value;
  public int getValue() {
   return value;
  }
  public void setValue(int value) {
   this.value = value;
  }
}

class Summation implements Runnable
{
  private int upper;
  private MutableInteger sumValue;
  public Summation(int upper, MutableInteger sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }
  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setValue(sum);
  }
}
```

```
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
     MutableInteger sum = new MutableInteger();
     int upper = Integer.parseInt(args[0]);
     Thread thrd = new Thread(new Summation(upper, sum));
     thrd.start();
     try {
       thrd.join();
       System.out.println
              ("The sum of "+upper+" is "+sum.getValue());
     } catch (InterruptedException ie) { }
    }
   }
   else
    System.err.println("Usage: Summation <integer value>");
   }
}
```

Another Example: Producer-Consumer

Define a class that implements **Runnable** interface

```
import java.util.Date;

class Producer implements Runnable
{
  private Channel<Date> queue;

  public Producer(Channel<Date> queue) {
    this.queue = queue;
  }

  public void run() {
    Date message;

    while (true) {
      // nap for awhile
      SleepUtilities.nap();

      // produce an item and enter it into the buffer
      message = new Date();
      System.out.println("Producer produced " + message);
      queue.send(message);
    }
  }
}
```

```
import java.util.Date;

public class Factory
{
   public static void main(String args[]) {
      // create the message queue
      Channel<Date> queue = new MessageQueue<Date>();

      // Create the producer and consumer threads and pass
      // each thread a reference to the MessageQueue object.
      Thread producer = new Thread(new Producer(queue));
      Thread consumer = new Thread(new Consumer(queue));

      // start the threads
      producer.start();
      consumer.start();
   }

}
```

```
      // consume an item from the buffer
      message = queue.receive();

      if (message != null)
        System.out.println("Consumer consumed " + message);
    }
  }
}
```

Java thread pool example

```
class Task implements Runnable
{
  public void run() {
   System.out.println("I am working on a task.");
  }
}
```

```
import java.util.concurrent.*;
public class TPExample
{
  public static void main(String[] args) {
     int numTasks = Integer.parseInt(args[0].trim());

     // create the thread pool
     ExecutorService pool =
           Executors.newCachedThreadPool();

     // run each task using a thread in the pool
     for (int i = 0; i < 5; i++)
         pool.execute(new Task());

     // sleep for 5 seconds
    try { Thread.sleep(5000); }
    catch (InterruptedException ie) { }
    pool.shutdown();
 }
}
```

- Work queue
  - Fixed number of threads

Possible problems

- Deadlock
- Resource thrashing
- Thread leakage
- Overload

*

Semantics of **fork()** and **exec()** system calls

Thread cancellation of target thread

Signal handling

Thread pools

Thread-specific data

Scheduler activations

OTHER THREADING ISSUES

Semantics of fork() and exec()

- What will happen if one thread in a process call **fork( )** to create a new process?
  - Does **fork()** duplicate only the calling thread or all threads?
  - How many threads in the new process?
- Duplicate only the invoking thread
  - exec(): will load another program
  - Everything will be replaced anyway
- Duplicate all threads
  - If exec() is not the next step after forking
  - What about threads performed blocking system call?!

Thread Cancellation

- Terminating a thread before it has finished
  - Examples
    - Threads search in parallel of database: one finds ☾ others stop
    - Stop fetching web contents (images)
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
    - - Thread resources and data consistency
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - + Wait for self cleanup ☾ **cancellation safety points**

	putpmsg	sigsuspend
	pwrite	sigtimedwait
	read	sigwait
	readv	sigwaitinfo
	recv	sleep
	recvfrom	system
	recvmsg	tcdrain
	select	usleep
	sem_timedwait	wait
wait	sem_wait	waitid
	send	waitpid
	sendmsg	write
	sendto	writev
	sigpause	

Thread Cancellation (cont.)

- After making the request
  - pthread_cancel returns
  - Does not mean the target thread has terminated
- Thread's and states
  - PTHREAD_CANCEL_ENABLE : default state
  - PTHREAD_CANCEL_DISABLE
  - A thread receives cancel request if it enters enable state
- A thread can change its state with

```
int pthread_setcancelstate(int state,
                           int  *oldstate);
```

Thread Cancellation (cont.)

- Possible cancellation types
  - PTHREAD_CANCEL_ASYNCHRONOUS: act on requests at any time
  - PTHREAD_CANCEL_DEFERRED: at on request only at cancellation points
  - set the cancellation type with `pthread_setcanceltype`

```
int pthread_setcanceltype(int type, int *oldtype);
```

- Cancellation points
  - Certain blocking functions cause cancellation points
  - Set a cancellation point with `pthread_testcancel`

```
void pthread_testcancel(void);
```

POSIX Functions: Cancellation Points

accept	mq_timedsend	putpmsg	sigsuspend
aio_suspend	msgrcv	pwrite	sigtimedwait
clock_nanosleep	msgsnd	read	sigwait
close	msync	readv	sigwaitinfo
connect	nanosleep	recv	sleep
creat	open	recvfrom	system
fcntl*	pause	recvmsg	tcdrain
fsync	poll	select	usleep
getmsg	pread	sem_timedwait	wait
getpmsg	pthread_cond_timedwait	sem_wait	waitid
lockf	pthread_cond_wait	send	waitpid
mq_receive	pthread_join	sendmsg	write
mq_send	pthread_testcancel	sendto	writev
mq_timedreceive	putmsg	sigpause	

These functions can always be cancellation points!
There are other functions that maybe or cannot be!

Signal Handling

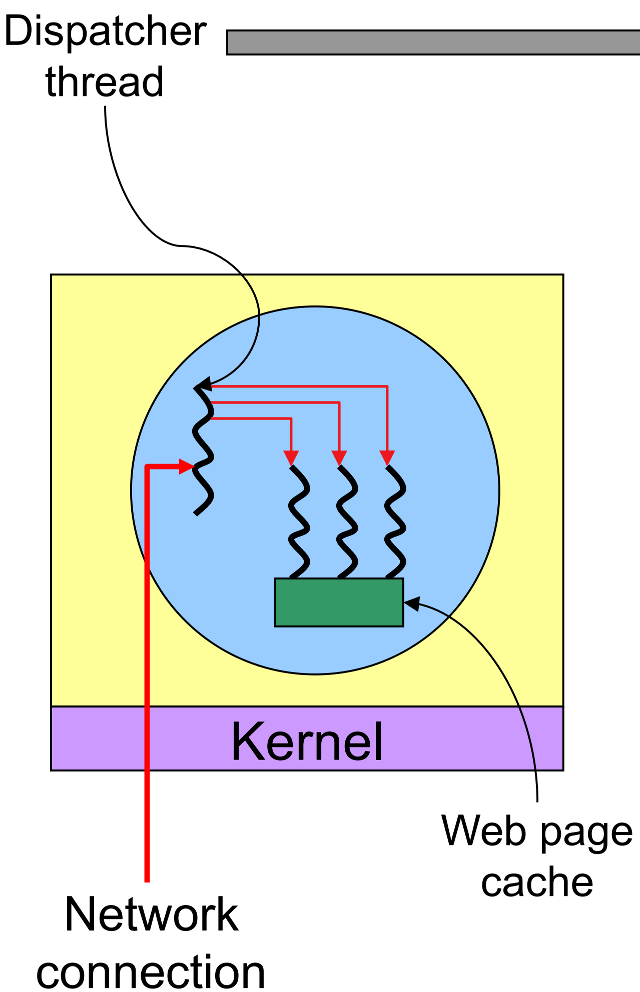
1. Signal is generated by particular event
2. Signal is delivered to a process
3. Signal is handled

- Signals are used in UNIX systems to notify a **process** that a particular event has occurred.
  - Depending on the source, we can classify them as
    - Synchronously [Running prog generates it] (e.g., div by 0, memory access)
    - Asynchronously [External src generates it] (e.g., ready of I/O or Ctrl+C)
- Which threads to notify?
  - All threads (Ctrl-C)
  - Single thread to which the signal applies (illegal memory, div by 0)
  - Subset of threads: thread set what it wants (**mask**)
  - **Thread handler: kernel default** or **user-defined**
- Unix allows threads to specify which one to block or accept
- Windows has no support for signals but it can be emulated

Thread Pool

- Recall web server example,
  - We created a thread for every request
  - This is better than creating a process, but still time consuming
  - No limit is put on the number of threads
- Pool of threads
  - Create some number of treads at the startup
  - These threads will wait to work and put back into pool
- 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
- Adjust thread number in pool
  - According to usage pattern and system load

Thread Pool Example: Web server




```
while(TRUE) {
  getNextRequest(&buf);
  handoffWork(&buf);
}
```

```
while(TRUE) {
  waitForWork(&buf);
  lookForPageInCache(&buf,&page);
  if(pageNotInCache(&page)) {
    readPageFromDisk(&buf,&page);
  }
  returnPage(&page);
}
```

Thread Specific Data

- Allows each thread to have its own copy of data
- We may not want to share all data
- Thread libraries have support for this
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

- Both M:M and Two-level models require **communication** to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
  - Events to invoke upcall
    - A thread make a blocking system calls
    - A blocking system call complete returns
    - To ask user-level thread scheduler (runtime systems) to select the next runnable thread
- This communication allows an application to maintain the correct number of kernel threads

SKIP THE REST

Windows XP Threads

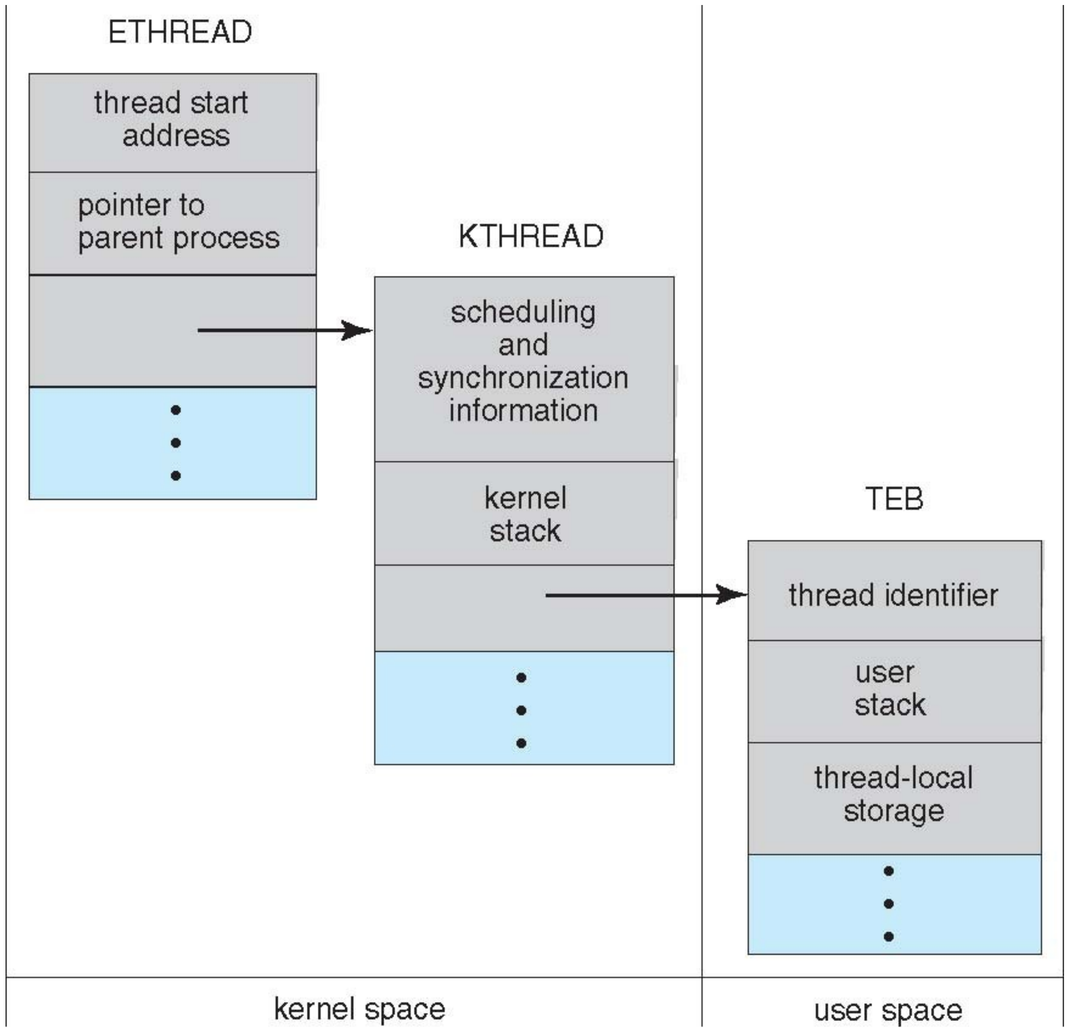
Linux Thread

OPERATING SYSTEM EXAMPLES

Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Windows XP Threads



*

Linux Threads

- Linux uses the term ***task*** (rather than process or thread) when referring to a flow of control
- Linux provides *clone()* system call to create threads
  - A set of flags, passed as arguments to the *clone()* system call determine how much sharing is involved (e.g. open files, memory space, etc.)
- Linux: 1-to-1 thread mapping
  - NPTL (Native POSIX Thread Library)

Linux Threads

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.