CS 332/532 Systems Programming

Lecture 30

Threads

Professor : Mahmut Unan – UAB CS

Agenda

- popen pclose functions
- Threads

popen and pclose functions

- As we have seen in the examples, the common usage of pipes involve creating a pipe, creating a child process with fork, closing the unused ends of the pipe, execing a command in the child process, and waiting for the child process to terminate in the parent process.
- Since this is such a common usage, UNIX systems provide *popen* and *pclose* functions that perform most of these operations in a single operation.
- The C APIs for the *popen* and *pclose* functions are shown below:

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- The popen function performs the following steps:
- creates a pipe
- creates a new process using fork
- perform the following steps in the child process
  - close unused ends of the pipe (based on the *type* argument)
  - execs a shell to execute the *command* provided as argument to popen (i.e., executes "sh -c command")
- perform the following steps in the parent process
  - close unused ends of the pipe (based on the *type* argument)
  - wait for the child process to terminate

- The *popen* function returns the FILE handle to the pipe created so that the calling process can read or write to the pipe using standard I/O system calls.
- If the *type* argument is specified as read-only ("r") then the calling process can read from the pipe, this results in reading from the *stdout* of the child process (see Figure 15.9).
- If the type argument is specifies as write-only ("w") then the calling process can write to the pipe, this results in writing to the *stdin* of the child process created (see Figure 15.10).

- The FILE handle returned by *popen* must be closed using *pclose* to make sure that the I/O stream opened to read or write to the pipe is closed and wait for the child process to terminate.
- The termination status of the shell started by exec will be returned when
the *pclose* function returns.
-

pipe2a.c

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

int main(int argc, char **argv) {
    FILE *fp1, *fp2;
    char line[BUFSIZ];

    if (argc != 3) {
        printf("Usage: %s <command1> <command2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* create a pipe, fork/exec command argv[1], in "read" mode */
    /* read mode - parent process reads stdout of child process */
    if ((fp1 = popen(argv[1], "r")) == NULL) {
    perror("popen");
    exit(EXIT_FAILURE);
    }

```

```c
    /* create a pipe, fork/exec command argv[2], in "write" mode */
    /* write mode - parent process writes to stdin of child process */
    if ((fp2 = popen(argv[2], "w")) == NULL) {
    perror("popen");
    exit(EXIT_FAILURE);
    }

    /* read stdout from child process 1 and write to stdin of
       child process 2 */
    while (fgets(line, BUFSIZ, fp1) != NULL) {
    if (fputs(line, fp2) == EOF) {
       printf("Error writing to pipe\n");
       exit(EXIT_FAILURE);
        }
    }

    /* wait for child process to terminate */
    if ((pclose(fp1) == -1) || pclose(fp2) == -1) {
    perror("pclose");
    exit(EXIT_FAILURE);
    }

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % ./pipe2a "ls -l" sort
-rw-r--r--@ 1 mahmutunan  staff    105 Nov  2 13:37 p1.c
-rw-r--r--@ 1 mahmutunan  staff    169 Nov  2 13:36 p2.c
-rw-r--r--@ 1 mahmutunan  staff    790 Oct 27 22:41 popen.c
-rw-r--r--@ 1 mahmutunan  staff   1694 Oct 27 22:41 pager2.c
-rw-r--r--@ 1 mahmutunan  staff   1853 Nov  4 13:07 pipe2a.c
-rw-r--r--@ 1 mahmutunan  staff   2073 Oct 27 22:41 pipe1.c
-rw-r--r--@ 1 mahmutunan  staff   2121 Oct 27 22:41 pipe0.c
-rw-r--r--@ 1 mahmutunan  staff   2284 Nov  4 12:44 pager.c
-rw-r--r--@ 1 mahmutunan  staff   2782 Nov  4 11:31 pipe2.c
-rw-r--r--@ 1 mahmutunan  staff   3858 Nov  4 11:51 pipe3.c
-rw-r--r--@ 1 mahmutunan  staff   5074 Nov  4 12:51 smalltale.txt
-rwxr-xr-x  1 mahmutunan  staff  12556 Nov  2 13:37 p1
-rwxr-xr-x  1 mahmutunan  staff  12604 Nov  2 13:37 p2
-rwxr-xr-x  1 mahmutunan  staff  12952 Nov  4 13:07 pipe2a
-rwxr-xr-x  1 mahmutunan  staff  12984 Nov  2 13:37 pipe0
-rwxr-xr-x  1 mahmutunan  staff  12996 Nov  2 13:20 pipe1
-rwxr-xr-x  1 mahmutunan  staff  13040 Nov  4 11:31 pipe2
-rwxr-xr-x  1 mahmutunan  staff  13040 Nov  4 12:41 pipe3
-rwxr-xr-x  1 mahmutunan  staff  13076 Nov  4 12:44 pager
total 352
```

- Note that since the command is executed using a shell, we can provide wildcards and other special characters that the shell can expand.
- Also note that in this version of the program the parent process is reading the *stdout* stream of the first child process and then writing to the *stdin* stream of the second child process (we did not do this in the first version).

Here is an updated version of the pager program that uses popen and pclose

pager2.c

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

int main(int argc, char **argv) {
  FILE *fpin, *fpout;
  char line[BUFSIZ];

  if (argc != 2) {
     printf("Usage: %s <filename>\n", argv[0]);
     exit(-1);
  }

  /* open file for reading */
  if ( (fpin = fopen( filename: argv[1],  mode: "r")) == NULL ) {
    printf("Error opening file %s for reading\n", argv[1]);
    exit(-1);
  }

  /* create a pipe, fork/exec process "more", in "write" mode */
  /* write mode - parent process writes, child process reads */
  if ( (fpout = popen("more", "w")) == NULL ) {
      perror("exec");
      exit(EXIT_FAILURE);
  }
```

```c
    /* read lines from the file and write it fpout */
    while (fgets(line, BUFSIZ, fpin) != NULL) {
        if (fputs(line, fpout) == EOF) {
            printf("Error writing to pipe\n");
        exit(EXIT_FAILURE);
        }
    }

    /* close the pipe and wait for child process to terminate */
    if (pclose(fpout) == -1) {
        perror("pclose");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pager2.c -o page
r2
(base) mahmutunan@MacBook-Pro lecture29 % ./pager2 smalltale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
we had everything before us we had nothing before us
we were all going direct to heaven we were all going direct
the other wayin short the period was so far like the present
```

popen.c

- You can also find a simpler version of the program that uses a single popen system call to create a pipe in "read" mode, execute the command specified as the command-line argument, reads the pipe and prints it to stdout

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

int main(int argc, char **argv) {
    FILE *fp;
    char line[BUFSIZ];

    if (argc != 2) {
        printf("Usage: %s <command>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

```

```
    if ((fp = popen(argv[1], "r")) == NULL) {
    perror("popen");
    exit(EXIT_FAILURE);
    }

    while (fgets(line, BUFSIZ, fp) != NULL) {
      fputs(line, stdout);
    }

    if (pclose(fp) == -1) {
    perror("pclose");
    exit(EXIT_FAILURE);
    }

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall popen.c -o popen

(base) mahmutunan@MacBook-Pro lecture29 % ./popen ps
  PID TTY           TIME CMD
 1600 ttys000    0:00.46 -zsh
26658 ttys000    0:00.00 ./popen ps
(base) mahmutunan@MacBook-Pro lecture29 % _
```

THREAD

Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
  - Suspending a process involves suspending all threads of the process
  - Termination of a process terminates all threads within the process

Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish

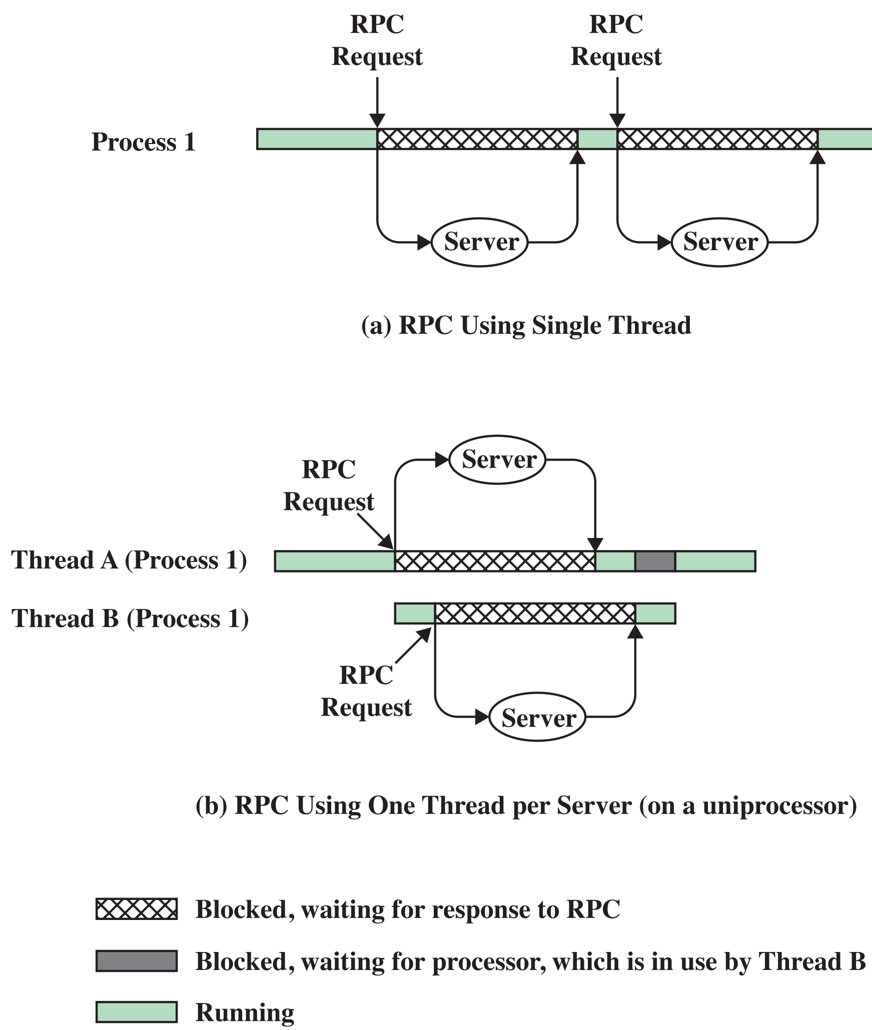


Figure 4.3 Remote Procedure Call (RPC) Using Threads

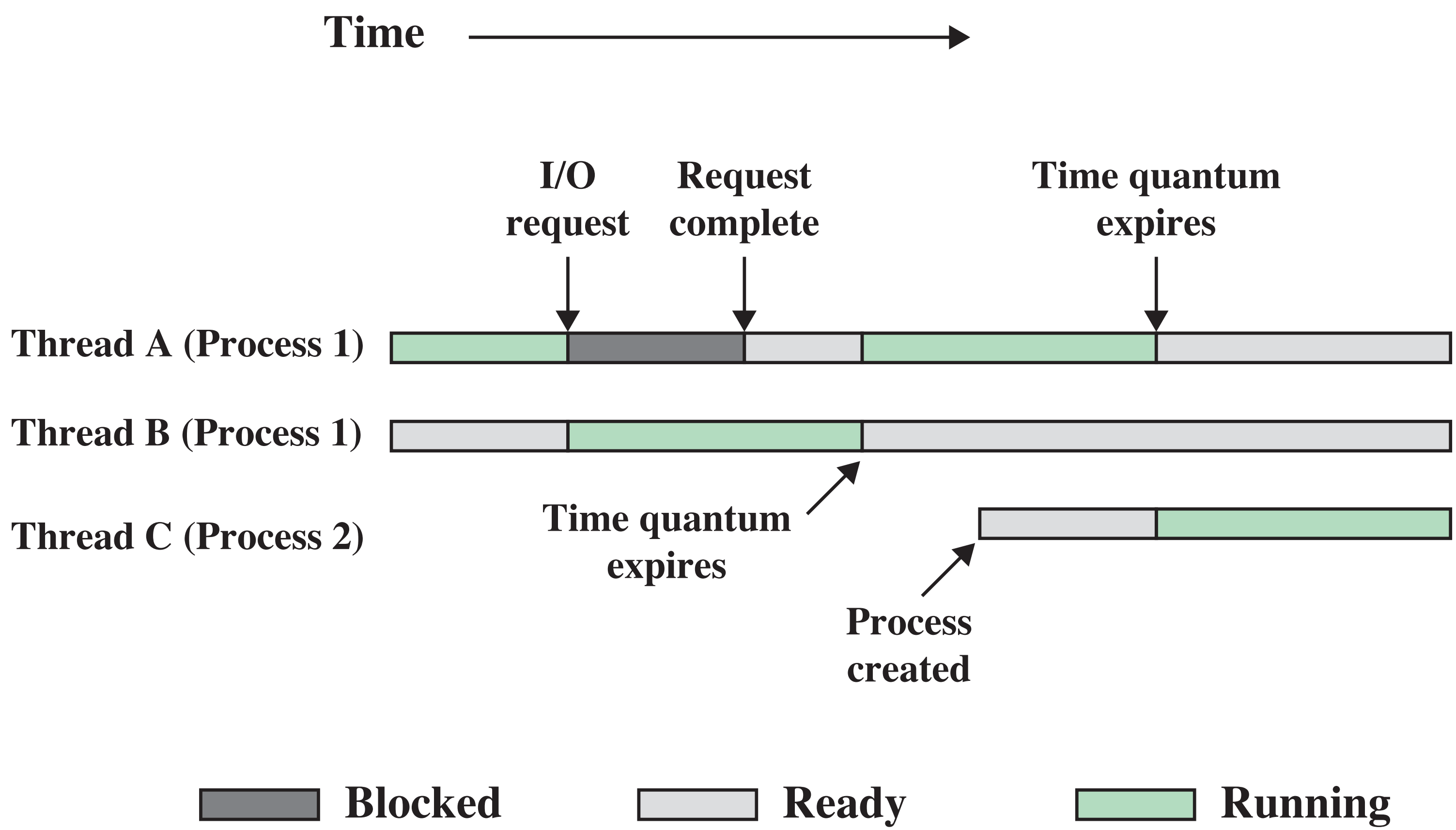
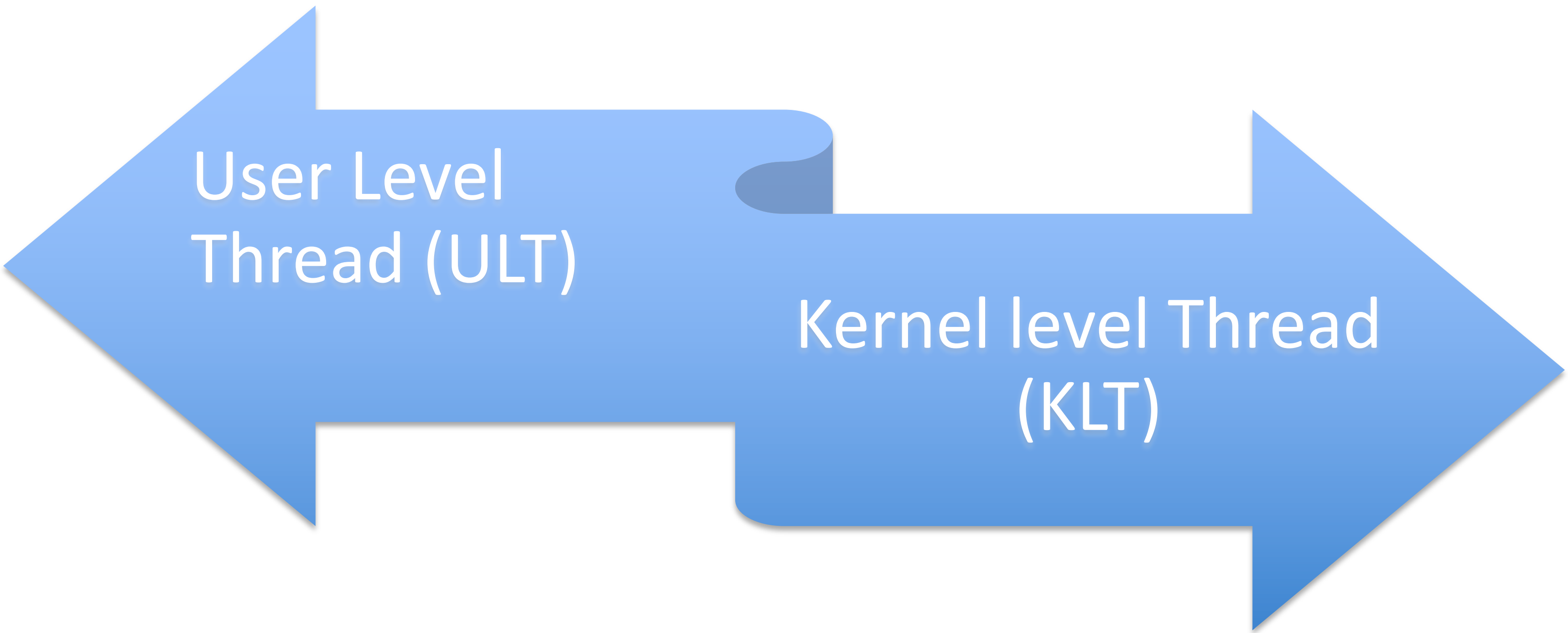


Figure 4.4 Multithreading Example on a Uniprocessor

Thread Synchronization

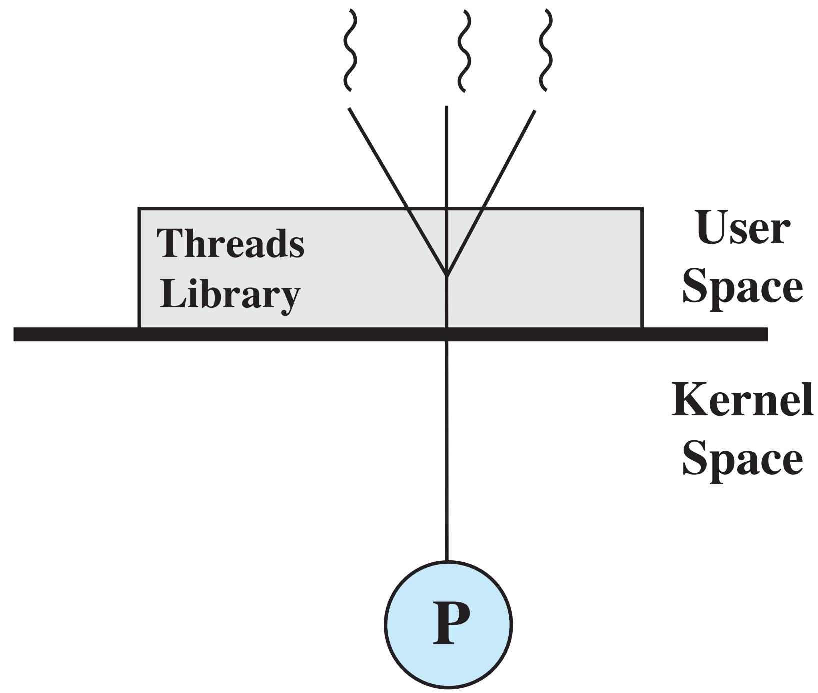
- It is necessary to synchronize the activities of the various threads
  - All threads of a process share the same address space and other resources
  - Any alteration of a resource by one thread affects the other threads in the same process

Types of Threads

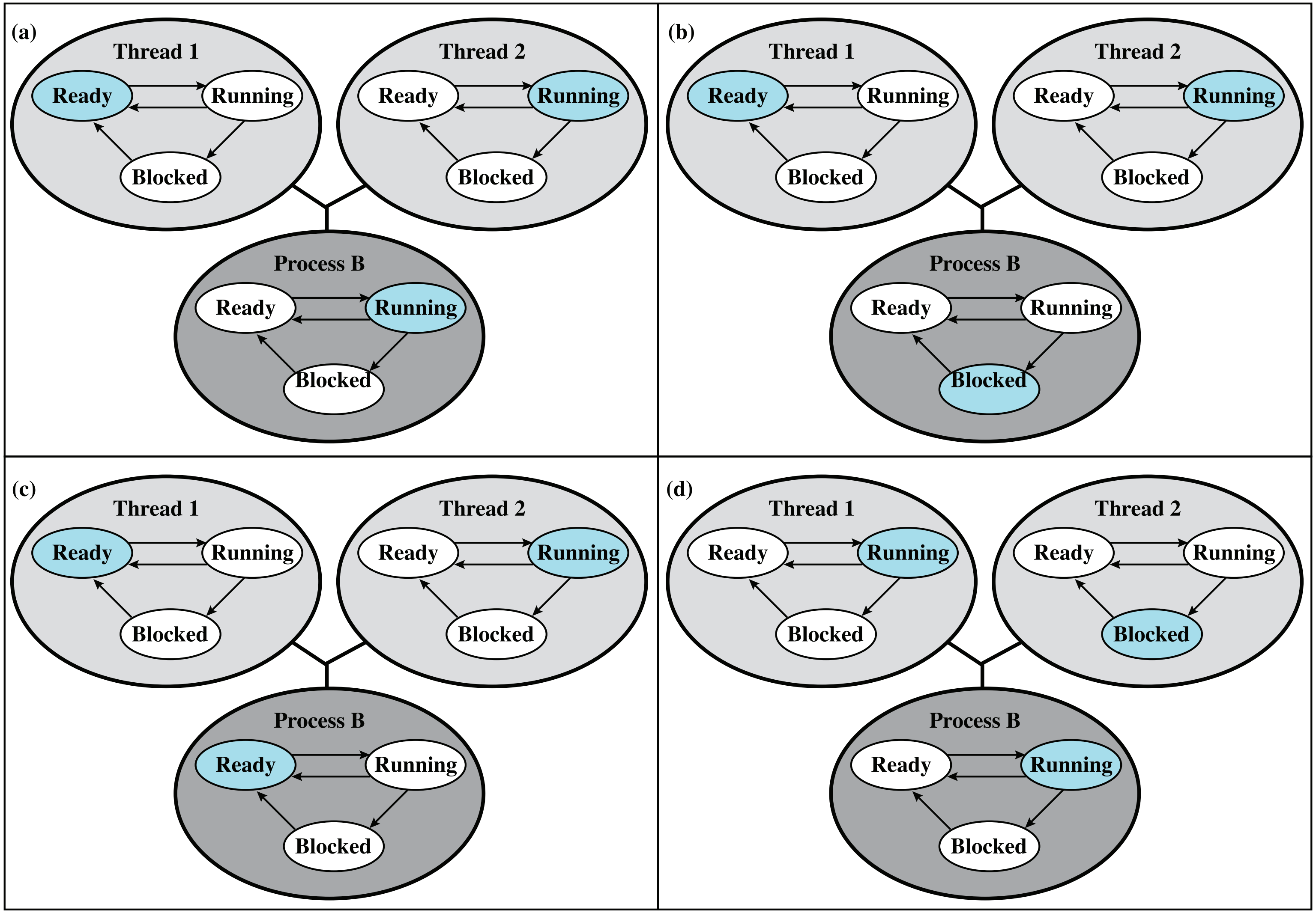


User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level



Colored state
is current state

Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

Advantages of ULTs

Thread switching does not require kernel mode privileges

Scheduling can be application specific

ULTs can run on any OS

Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - As a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
  - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time

Overcoming ULT Disadvantages

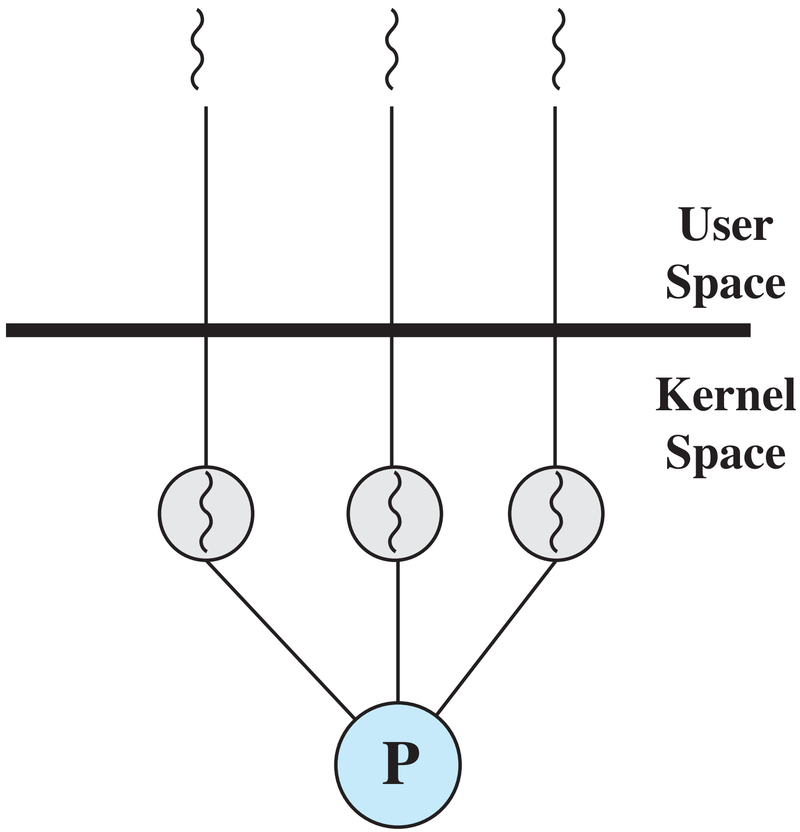
Jacketing

- Purpose is to convert a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads

- However, this approach eliminates the main advantage of threads

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the kernel
  - There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
  - Windows is an example of this approach

Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded

Disadvantage of KLTs

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1
Thread and Process Operation Latencies (µs)

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

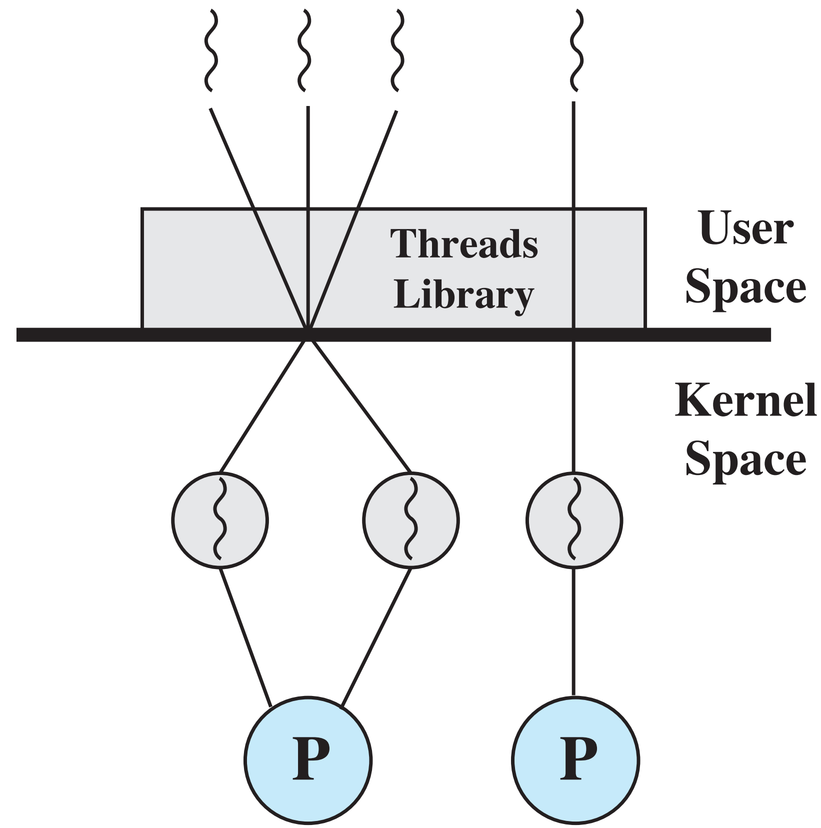
https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

Multithreading Models

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Combined Approaches

- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application
- Solaris is a good example



(c) Combined

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Table 4.2
Relationship between Threads and Processes

Create threads using POSIX threads library

- In the previous lectures/labs we focused on how to create processes, in this lab we will focus on creating threads and mechanisms for establishing synchronization among threads.
- First, let us understand the difference between a process and a thread.
  - A process could be considered to have two characteristics:
    - (a) resource ownership
    - (b) scheduling or execution.
- The unit of scheduling and dispatching is usually referred to as a **thread** or **lightweight process and** the ability of to support multiple, concurrent paths of execution within a single process is often referred to as *multithreading*.

- Threads offer several benefits compared to a process:
  - Threads takes less time to create a new thread than a process
  - Threads take less time to terminate a thread than a process
  - Switching between two threads (context switching) takes less time than switching between processes
  - All of the threads in a process share the state and resources of that process (since threads reside in the same address space and have access to the same data)
  - Threads enhance efficiency in communication between programs (since threads share memory and files within the same process and can communicate without invoking the kernel)

- As a result of these advantages, if we have to implement a set of functions that are closely related, implementing this functionality using multiple threads is far more efficient than using multiple processes.
- We will use the **POSIX threads library**, usually referred to as Pthreads library, that provides C APIs to create and manage threads. We have to include the file *pthread.h* and link with *-lpthread* to compile and link.
- We can create new threads using the *pthread_create()* function which has the following function definition:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const
pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

- The new thread that will be created by the *pthread_create* function will invoke the function *start_routine*.
- Note that the function *start_routine* takes one argument of type *void ** and has the return type as `void *`.
- In other words, the function *start_routine* has the following function definition:

```
void *start_routine(void *arg)
```

- When the *pthread_create* call returns successfully, it returns the thread ID associated with the new thread created in the variable thread.
- This can be used by the main thread in subsequent pthread function calls such as *pthread_join*.
- The second argument, *attr,* provides a reference to the *pthread_attr_t* structure that describes the various attributes of the new thread to be created.
- It can be initialized using *pthread_attr_init* call or set to NULL if default attributes must be used.
- You can find out more about the different thread attributes that can be specified by looking at the man page for *pthread_attr_init*.

- The new thread created will terminate when the function *start_routine* returns or when a call to *pthread_exit* is made inside the *start_routine*.
- We can use the *pthread_join* function to wait for a thread to complete using the thread ID that was returned when *pthread_create* call was invoked.
- If a thread has already completed,
  - *pthread_join* will return immediately, otherwise, it will wait for the corresponding thread to complete.

exercise 1

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


void *someFuncToCreateThread(void *someValue)
{
    sleep(2);
    printf("I am inside the thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("I am inside the main function\n");
    pthread_create(&thread_id, NULL, someFuncToCreateThread, NULL);
    pthread_join(thread_id, NULL);
    printf("Back to the main function\n");
    exit(0);
}

```

compile & run

To compile a multithreaded program, we will be using gcc and we need to link it with the pthreads library.

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc exercise1.c -o exercise1 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise1
I am inside the main function
I am inside the thread
Back to the main function
(base) mahmutunan@MacBook-Pro lecture31 %
```

exercise 2

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

void *function1(void *someValue)
{
    while(1==1) {
        sleep(1);
        printf("function 1 \n");
    }
}
void function2()
{
    while(1==1) {
        sleep(2);
        printf("function 2\n");
    }
}

int main()
{
    pthread_t thread_id;
    printf("I am inside the main function\n");
    pthread_create(&thread_id, NULL, function1, NULL);
    function2();
    exit(0);
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc exercise2.c -o exercise2 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise2
I am inside the main function
function 1
function 2
function 1
function 1
function 2
function 1
function 1
function 2
function 1
function 1
function 2
function 1
function 1
```

exercise 3

```c
int globalVar = 50; //define a global variable

void *someFuncToCreateThread(void *someValue)
{
    int *threadId = (int *)someValue; // Store the value argument passed to this thread

    //define a static and a local variable
    static int staticVar = 75;
    int localVar = 10;

    // let's change the variables
    globalVar +=100;
    staticVar +=100;
    localVar +=100;
    printf("id =%d,global = %d,  local = %d, static =%d, \n", *threadId, globalVar,localVar ,staticVar);

    return NULL;
}

int main()
{
    int i;
    pthread_t thread_id;
    for (i = 0; i < 4; i++)
        pthread_create(&thread_id, NULL, someFuncToCreateThread, (void *)&thread_id);
    pthread_exit(NULL);

}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc exercise3.c -o exercise3 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise3
id =151261184,global = 150,  local = 110, static =175,
id =151261184,global = 150,  local = 110, static =175,
id =151261184,global = 250,  local = 110, static =275,
id =151261184,global = 250,  local = 110, static =275,
(base) mahmutunan@MacBook-Pro lecture31 % _
```

Remember, global and static variables are stored in data segment.
All threads share data segment, so they are shared by all threads.

pthread1.c

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

int nthreads;

void *compute(void *arg) {
  long tid = (long)arg;

  printf("Hello, I am thread %ld of %d\n", tid, nthreads);

  return (NULL);
}

int main(int argc, char **argv) {
  long i;
  pthread_t *tid;

  if (argc != 2) {
    printf("Usage: %s <# of threads>\n",argv[0]);
    exit(-1);
  }

  nthreads = atoi(argv[1]); // no. of threads
```

```c

    // allocate vector and initialize
    tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);

    // create threads
    for ( i = 0; i < nthreads; i++)
      pthread_create(&tid[i], NULL, compute, (void *)i);

    // wait for them to complete
    for ( i = 0; i < nthreads; i++)
      pthread_join(tid[i], NULL);

    printf("Exiting main program\n");

    return 0;
}

```

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc pthread1.c -o exercise4 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise4 4
Hello, I am thread 0 of 4
Hello, I am thread 1 of 4
Hello, I am thread 2 of 4
Hello, I am thread 3 of 4
Exiting main program
```

pthread2.c

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

int nthreads;

void *compute(void *arg) {
  long tid = (long)arg;
  pthread_t pthread_id = pthread_self();

  printf("Hello, I am thread %ld of %d, pthread_self() = %lu (0x%lx)\n",
         tid, nthreads, (unsigned long)pthread_id, (unsigned long)pthread_id);

  return (NULL);
}

int main(int argc, char **argv) {
  long i;
  pthread_t *tid;
  pthread_t pthread_id = pthread_self();
```

```c

    if (argc != 2) {
      printf("Usage: %s <# of threads>\n",argv[0]);
      exit(-1);
    }

    nthreads = atoi(argv[1]); // no. of threads

    // allocate vector and initialize
    tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);

    // create threads
    for ( i = 0; i < nthreads; i++)
      pthread_create(&tid[i], NULL, compute, (void *)i);

    for ( i = 0; i < nthreads; i++)
      printf("tid[%ld] = %lu (0x%lx)\n", i, tid[i], tid[i]);

    printf("Hello, I am main thread. pthread_self() = %lu (0x%lx)\n",
           (unsigned long)pthread_id, (unsigned long)pthread_id);

    // wait for them to complete
    for ( i = 0; i < nthreads; i++)
      pthread_join(tid[i], NULL);

    printf("Exiting main program\n");

    return 0;
}
```

```
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise5 4
tid[0] = 123145541038080 (0x70000e3ab000)
tid[1] = 123145541574656 (0x70000e42e000)
tid[2] = 123145542111232 (0x70000e4b1000)
tid[3] = 123145542647808 (0x70000e534000)
Hello, I am main thread. pthread_self() = 4365594048 (0x10435adc0)
Hello, I am thread 1 of 4, pthread_self() = 123145541574656 (0x70000e42e000)
Hello, I am thread 2 of 4, pthread_self() = 123145542111232 (0x70000e4b1000)
Hello, I am thread 0 of 4, pthread_self() = 123145541038080 (0x70000e3ab000)
Hello, I am thread 3 of 4, pthread_self() = 123145542647808 (0x70000e534000)
Exiting main program
```

pthread3.c

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

typedef struct foo {
   pthread_t ptid;  /* thread id returned by pthread_create */
   int tid;         /* user managed thread id (0 through nthreads-1) */
   int nthreads;    /* total no. of threads created */
} FOO;

void *compute(void *args) {
  FOO *info = (FOO *)args;
  printf("Hello, I am thread %d of %d\n", info->tid, info->nthreads);

  return (NULL);
}

int main(int argc, char **argv) {
  int i, nthreads;
  FOO *info;

  if (argc != 2) {
    printf("Usage: %s <# of threads>\n",argv[0]);
    exit(-1);
  }
```

```c
    nthreads = atoi(argv[1]); // no. of threads

    // allocate structure
    info = (FOO *)malloc(sizeof(FOO)*nthreads);

    // create threads
    for ( i = 0; i < nthreads; i++) {
        info[i].tid = i;
        info[i].nthreads = nthreads;
        pthread_create(&info[i].ptid, NULL, compute, (void *)&info[i]);
    }

    // wait for them to complete
    for ( i = 0; i < nthreads; i++)
        pthread_join(info[i].ptid, NULL);

    free(info);
    printf("Exiting main program\n");

    return 0;
}
```

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc pthread3.c -o exercise6 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise6 4
Hello, I am thread 1 of 4
Hello, I am thread 0 of 4
Hello, I am thread 2 of 4
Hello, I am thread 3 of 4
Exiting main program
(base) mahmutunan@MacBook-Pro lecture31 %
```