

CS 332/532 Systems Programming

Lecture 28
Pipes / 2

Professor : Mahmut Unan – UAB CS

Agenda

- Pipes continued
- popen
- Pclose

Thread

Exercise 2

In this example, we will use the first option

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6  #include <sys/stat.h>
7
8  int main(int argc, char **argv) {
9      pid_t pid;
10     int pipefd[2]; /* fildes[0] for read, fildes[1] for write */
11
12     if (argc != 3) {
13         printf("Usage: %s <command1> <command2>\n", argv[0]);
14         exit(EXIT_FAILURE);
15     }
16
17     if (pipe(pipefd) == 0) { /* Open a pipe */
18         pid = fork(); /* fork child process to execute command2 */
19         if (pid == 0) { /* this is the child process */
20             /* close write end of the pipe */
21             close(pipefd[1]);
22
23             /* replace stdin with read end of pipe */
24             if (dup2(pipefd[0], 0) == -1) {
25                 perror("dup2");
26                 exit(EXIT_FAILURE);
27             }
```

Exercise 2

```
28
29     /* execute <command2> */
30     execlp(argv[2], argv[2], (char *)NULL);
31     perror("execlp");
32     exit(EXIT_FAILURE);
33 } else if (pid > 0) { /* this is the parent process */
34     /* close read end of the pipe */
35     close(pipefd[0]);
36
37     /* replace stdout with write end of pipe */
38     if (dup2(pipefd[1], 1) == -1) {
39         perror("dup2");
40         exit(EXIT_FAILURE);
41     }
42
43     /* execute <command1> */
44     execlp(argv[1], argv[1], (char *)NULL);
45     perror("execlp");
46     exit(EXIT_FAILURE);
47 } else if (pid < 0) { /* we have an error */
48     perror("fork"); /* use perror to print the system error message */
49     exit(EXIT_FAILURE);
50 }
51 } else {
52     perror("pipe");
53     exit(EXIT_FAILURE);
54 }
55
56 return 0;
57 }
```

compile & run

```
pipe1.c × pipe0.c × p2.c × p1.c ×
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     int a = 15, b = 25;
5     printf("%d\n", a+b);
6
7 }
```

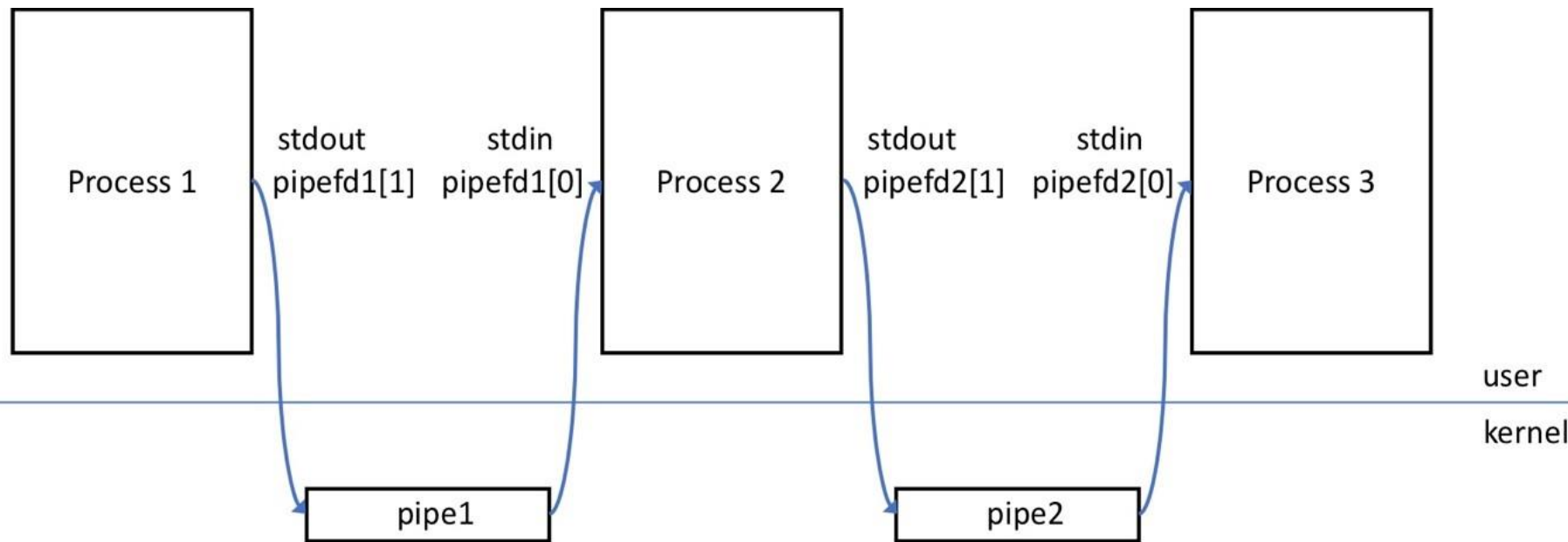
```
pipe1.c × pipe0.c × p2.c × p1.c ×
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     int x;
5     printf("Enter an int \n");
6     scanf("%d",&x);
7     printf("square of your number is %d \n", x*x);
8
9 }
```

```
(base) mahmutunan@MacBook-Pro lecture28 % gcc p1.c -o p1
(base) mahmutunan@MacBook-Pro lecture28 % ./p1
40
(base) mahmutunan@MacBook-Pro lecture28 % gcc p2.c -o p2
(base) mahmutunan@MacBook-Pro lecture28 % ./p2
Enter an int
5
square of your number is 25
(base) mahmutunan@MacBook-Pro lecture28 % gcc pipe0.c -o pipe0
(base) mahmutunan@MacBook-Pro lecture28 % ./pipe0 ./p1 ./p2
Enter an int
square of your number is 1600
(base) mahmutunan@MacBook-Pro lecture28 %
```

extend to three process

- We can extend this to three processes if we like to implement something like: *ls / sort / wc*.
- We will create three processes and use two pipes - one for communication between the first and second process and one for communication between second and third process.
- The code for the first and third children will be similar to the example above while the second child has to replace both standard input and standard output streams instead of just one. T

The diagram below illustrates this



pipe3.c

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <sys/wait.h>
7 #include <sys/stat.h>
8
9 int main(int argc, char **argv) {
10     pid_t pid1, pid2, pid3;
11     int pipefd1[2]; /* pipefd1[0] for read, pipefd1[1] for write */
12     int pipefd2[2]; /* pipefd2[0] for read, pipefd2[1] for write */
13     int status1, status2, status3;
14
15     if (argc != 4) {
16         printf("Usage: %s <command1> <command2> <command3>\n", argv[0]);
17         exit(EXIT_FAILURE);
18     }
19
20     if (pipe(pipefd1) != 0) { /* Open pipefd1 */
21         perror("pipe");
22         exit(EXIT_FAILURE);
23     }
24
25     if (pipe(pipefd2) != 0) { /* Open pipefd2 */
26         perror("pipe");
27         exit(EXIT_FAILURE);
28     }
29
30     pid1 = fork(); /* fork first process to execute command1 */
31     if (pid1 == 0) { /* this is the child process */
32         /* close read end of the pipefd1 */
33         close(pipefd1[0]);
34
35         /* close both ends of pipefd2 */
36         close(pipefd2[0]);
37         close(pipefd2[1]);
```



```

39      /* replace stdout with write end of pipefd1 */
40      if (dup2(pipefd1[1], 1) == -1) {
41          perror("dup2");
42          exit(EXIT_FAILURE);
43      }
44
45      /* execute <command1> */
46      execlp( file: argv[1], arg0: argv[1], (char *)NULL);
47      perror("execlp");
48      exit(EXIT_FAILURE);
49
50  } else if (pid1 < 0) { /* we have an error */
51      perror("fork"); /* use perror to print the system error message */
52      exit(EXIT_FAILURE);
53  }
54
55  pid2 = fork(); /* fork second process to execute command2 */
56  if (pid2 == 0) { /* this is child process */
57      /* close write end of the pipefd1 */
58      close(pipefd1[1]);
59
60      /* replace stdin with read end of pipefd2 */
61      if (dup2(pipefd1[0], 0) == -1) {
62          perror("dup2");
63          exit(EXIT_FAILURE);
64      }
65
66      /* close read end of pipefd2 */
67      close(pipefd2[0]);
68
69      /* replace stdout with write end of pipefd2 */
70      if (dup2(pipefd2[1], 1) == -1) {
71          perror("dup2");
72          exit(EXIT_FAILURE);
73      }
74
75      /* execute <command2> */
76      execlp( file: argv[2], arg0: argv[2], (char *)NULL);
77      perror("execlp");
78      exit(EXIT_FAILURE);

```

```

79
80 } else if (pid2 < 0) { /* we have an error */
81     perror("fork"); /* use perror to print the system error message */
82     exit(EXIT_FAILURE);
83 }
84
85 pid3 = fork(); /* fork third process to execute command3 */
86 if (pid3 == 0) { /* this is child process */
87     /* close both ends of the pipefd1 */
88     close(pipefd1[0]);
89     close(pipefd1[1]);
90
91     /* close write end of pipefd2 */
92     close(pipefd2[1]);
93
94     /* replace stdin with read end of pipefd2 */
95     if (dup2(pipefd2[0], 0) == -1) {
96         perror("dup2");
97         exit(EXIT_FAILURE);
98     }
99
100     /* execute <command3> */
101     execlp( file: argv[3], arg0: argv[3], (char *)NULL);
102     perror("execlp");
103     exit(EXIT_FAILURE);
104
105 } else if (pid3 < 0) { /* we have an error */
106     perror("fork"); /* use perror to print the system error message */
107     exit(EXIT_FAILURE);
108 }
109
110 /* close the pipes in the parent */
111 close(pipefd1[0]);
112 close(pipefd1[1]);
113 close(pipefd2[0]);
114 close(pipefd2[1]);
115
116 /* wait for both child processes to terminate */
117 waitpid(pid1, &status1, 0);
118 waitpid(pid2, &status2, 0);
119 waitpid(pid3, &status3, 0);
120
121 return 0;
122 }

```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pipe3.c -o pipe3
```

```
(base) mahmutunan@MacBook-Pro lecture29 % ./pipe3 ps sort wc
```

```
5      23     161
```

```
(base) mahmutunan@MacBook-Pro lecture29 % _
```

pager.c

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <string.h>
6 #include <ctype.h>
7 #include <sys/wait.h>
8 #include <sys/stat.h>
9
10 int main(int argc, char **argv) {
11     pid_t pid;
12     int status;
13     int pipefd[2]; /* pipefd[0] for read, pipefd[1] for write */
14     FILE *fp;
15     char line[BUFSIZ];
16     int n;
17
18     if (argc != 2) {
19         printf("Usage: %s <filename>\n", argv[0]);
20         exit(-1);
21     }
22
23     if ( (fp = fopen(argv[1], "r")) == NULL) {
24         printf("Error opening file %s for reading\n", argv[1]);
25         exit(-1);
26     }
27
28     if (pipe(pipefd) == 0) { /* Open a pipe */
29         if ((pid = fork()) == 0) { /* I am the child process */
30             close(pipefd[1]); /* close write end */
31             dup2(pipefd[0], STDIN_FILENO); /* replace stdin of child */
32             execlp("/usr/bin/more", "more", (char *)NULL);
33             perror("exec");
34             exit(EXIT_FAILURE);
35         }
```

```

34     exit(EXIT_FAILURE);
35 } else if (pid > 0) { /* I am the parent process */
36     close(pipefd[0]); /* close read end */
37     /* read lines from the file and write it to pipe */
38     while (fgets(line, BUFSIZ, fp) != NULL) {
39         n = strlen(line);
40         if (write(pipefd[1], line, n) != n) {
41             printf("Error writing to pipe\n");
42             exit(-1);
43         }
44     }
45     close(pipefd[1]); /* close write end */
46
47     wait(&status); /* wait for child to terminate */
48     if (WIFEXITED(status))
49         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
50     else
51         printf("Child process did not terminate normally!\n");
52 } else { /* we have an error in fork */
53     perror("fork");
54     exit(EXIT_FAILURE);
55 }
56 } else {
57     perror("pipe");
58     exit(EXIT_FAILURE);
59 }
60
61 exit(EXIT_SUCCESS);
62 }

```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % ./pager 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 way in short the period was so far like the present  
period that some of its noisiest authorities insisted on its  
being received for good or for evil in the superlative degree  
of comparison only
```

```
there were a king with a large jaw and a queen with a plain face  
on the throne of england there were a king with a large jaw and  
a queen with a fair face on the throne of france in both  
countries it was clearer than crystal to the lords of the state
```

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char **argv) {
5      FILE *fp1, *fp2;
6      char line[BUFSIZ];
7
8      if (argc != 3) {
9          printf("Usage: %s <command1> <command2>\n", argv[0]);
10         exit(EXIT_FAILURE);
11     }
12
13     /* create a pipe, fork/exec command argv[1], in "read" mode */
14     /* read mode - parent process reads stdout of child process */
15     if ((fp1 = popen(argv[1], "r")) == NULL) {
16         perror("popen");
17         exit(EXIT_FAILURE);
18     }
19 }
```

```
19
20     /* create a pipe, fork/exec command argv[2], in "write" mode */
21     /* write mode - parent process writes to stdin of child process */
22     if ((fp2 = popen(argv[2], "w")) == NULL) {
23         perror("popen");
24         exit(EXIT_FAILURE);
25     }
26
27     /* read stdout from child process 1 and write to stdin of
28        child process 2 */
29     while (fgets(line, BUFSIZ, fp1) != NULL) {
30         if (fputs(line, fp2) == EOF) {
31             printf("Error writing to pipe\n");
32             exit(EXIT_FAILURE);
33         }
34     }
35
36     /* wait for child process to terminate */
37     if ((pclose(fp1) == -1) || pclose(fp2) == -1) {
38         perror("pclose");
39         exit(EXIT_FAILURE);
40     }
41
42     return 0;
43 }
```

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).

pager2.c

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

```
#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);
    }
}
```

```
/* 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 pager2
(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 way in 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

```
#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

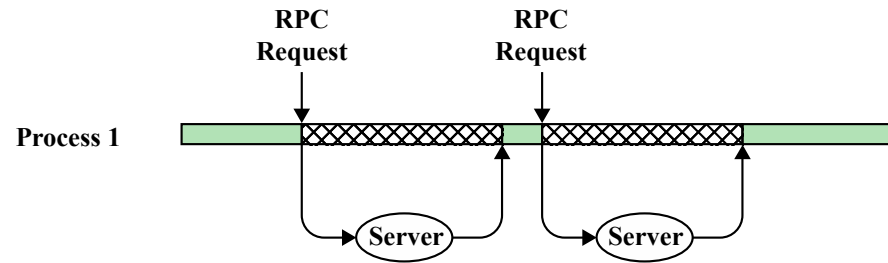
Thread Execution States

The key states for a thread are:

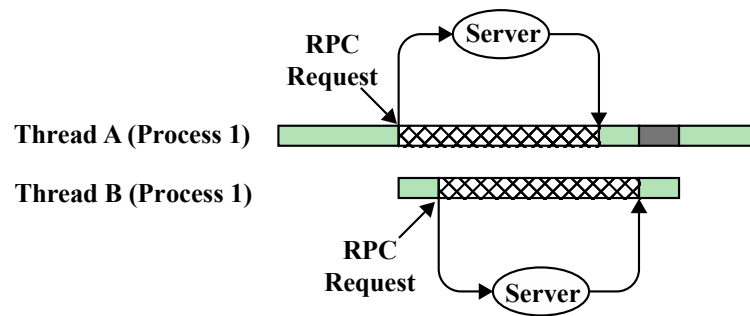
- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish



(a) RPC Using Single Thread



(b) RPC Using One Thread per Server (on a uniprocessor)

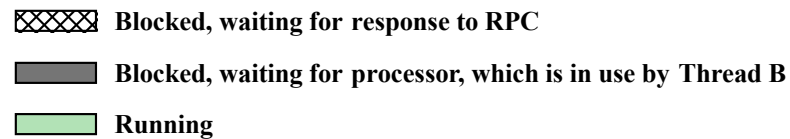


Figure 4.3 Remote Procedure Call (RPC) Using Threads

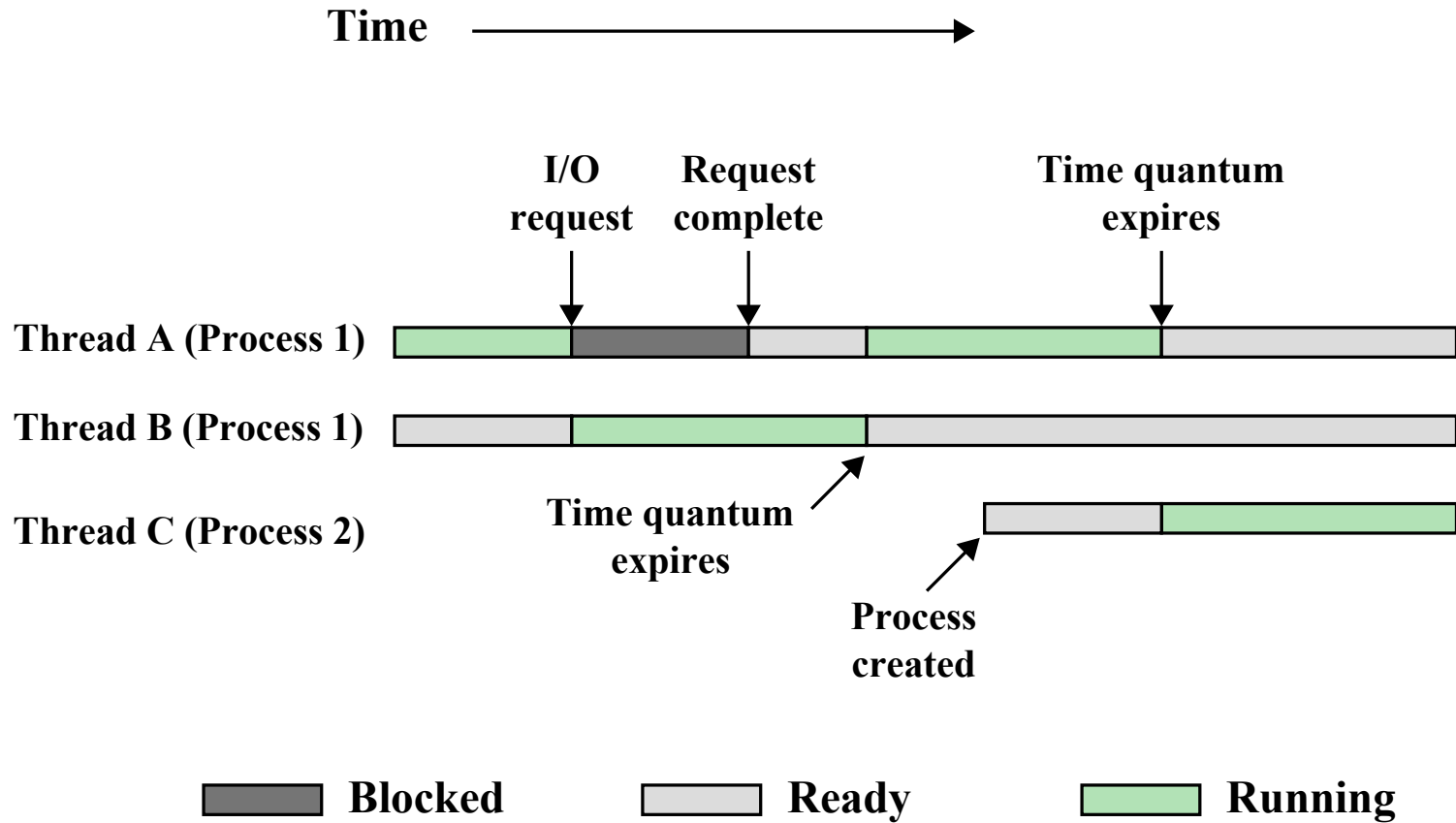
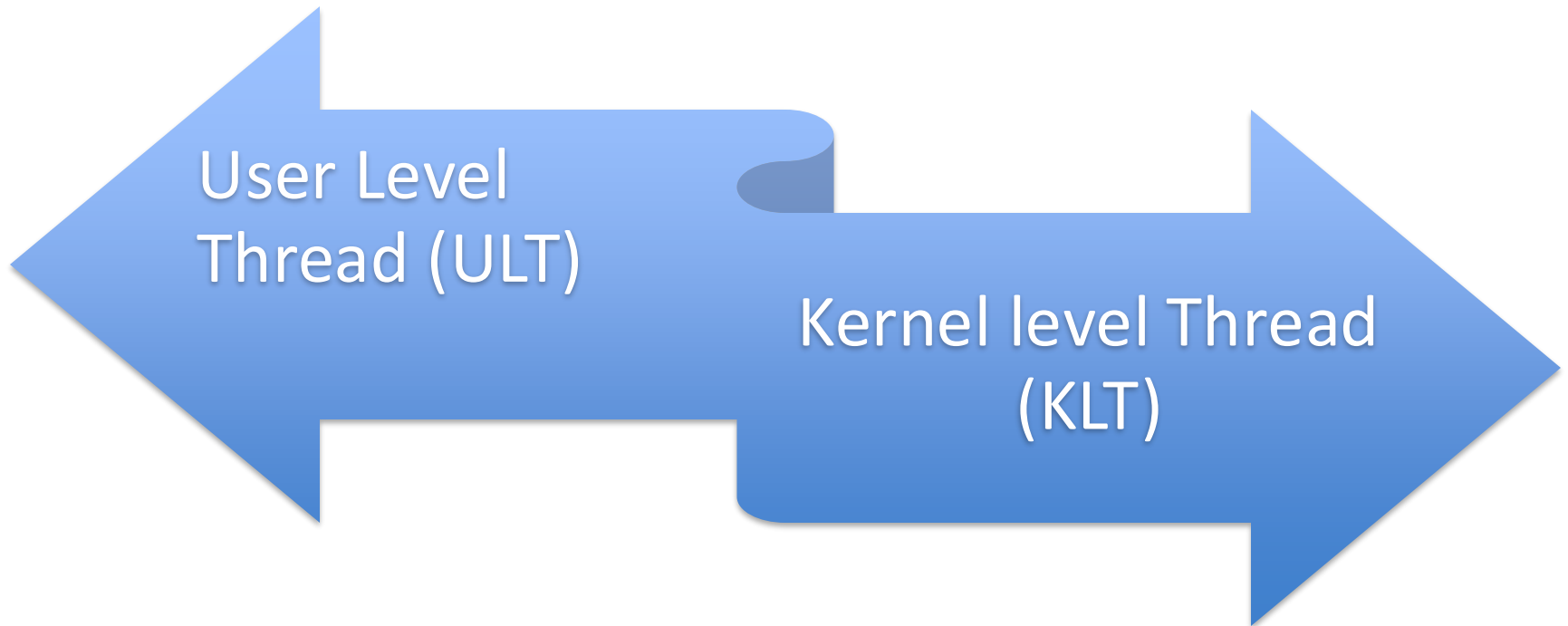


Figure 4.4 Multithreading Example on a Uniprocessor

Thread Synchronization

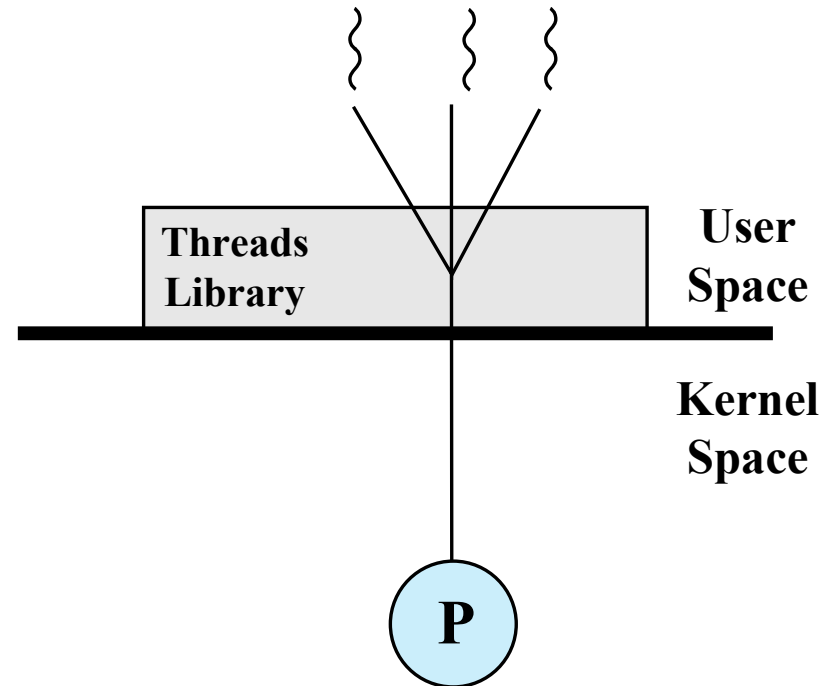
- It is necessary to synchronize the activities of the various threads
 - All threads of a process share the same address space and other resources
 - Any alteration of a resource by one thread affects the other threads in the same process

Types of Threads

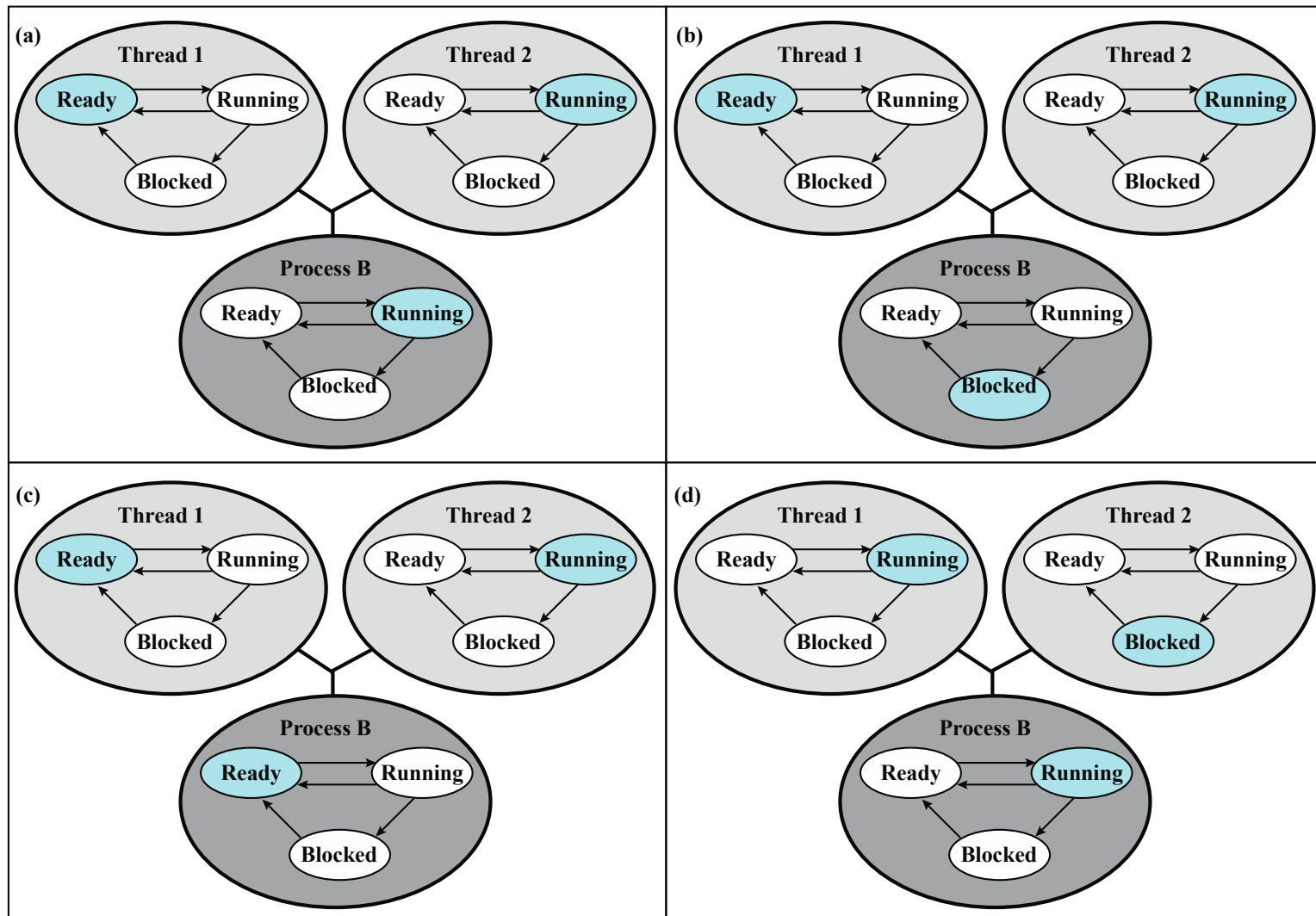


User-Level Threads (ULTs)

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



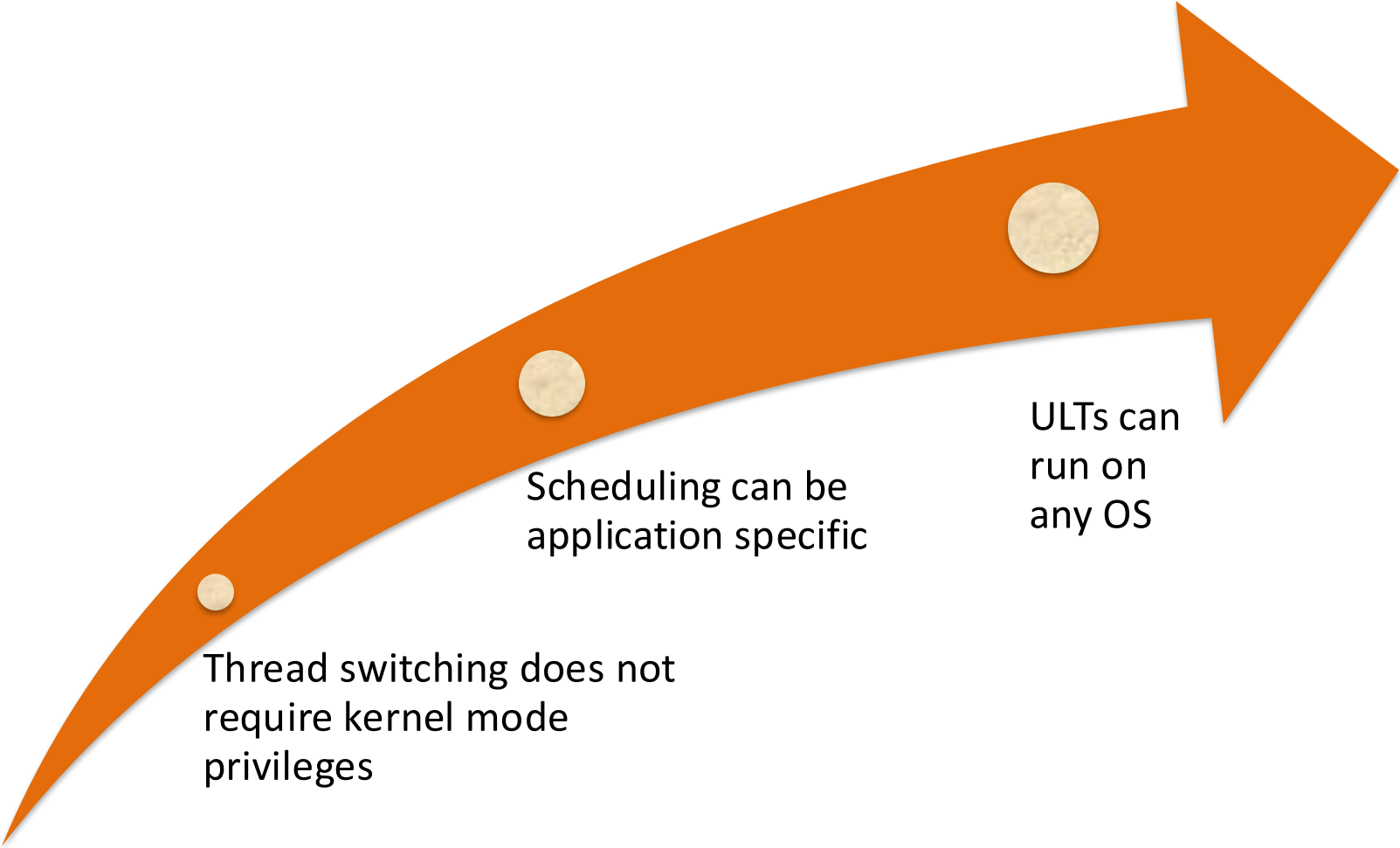
(a) Pure user-level



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

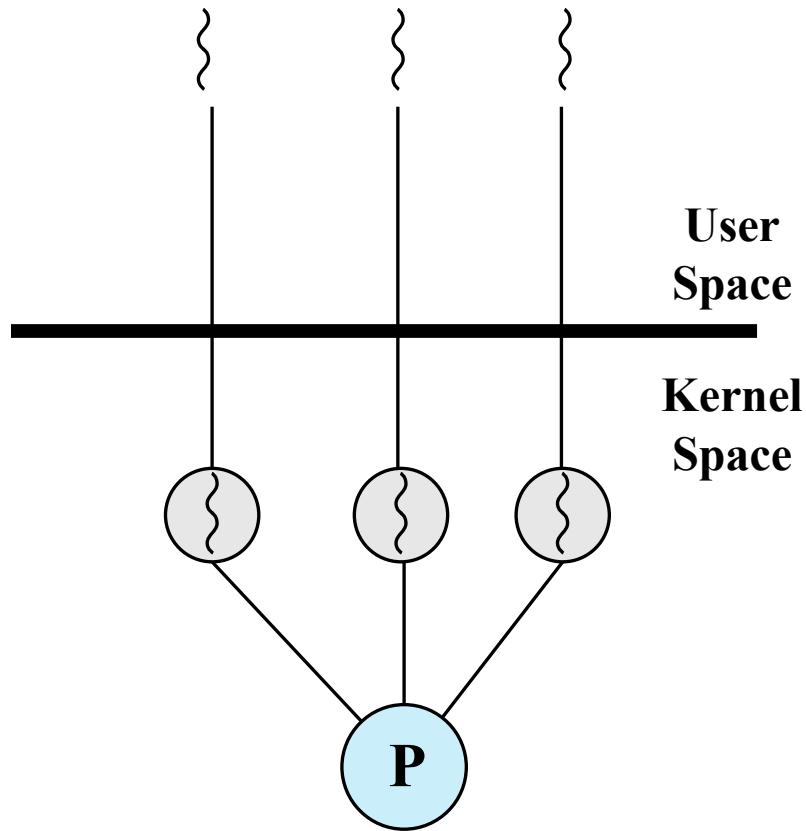
- Purpose is to convert a blocking system call into a non-blocking system call



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 (μ s)

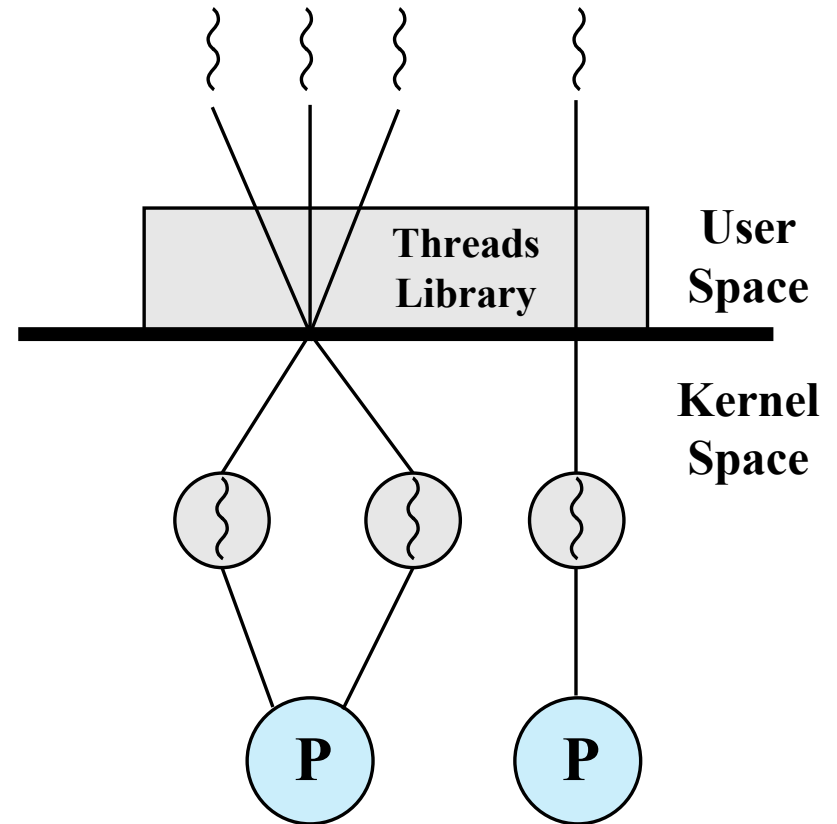
S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can 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.

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