Operating Systems

0107451

Chapter 2 Processes and Threads

Dr. Naeem Odat

College of Engineering
Department of Computer and Communications Engineering

2.2 Threads

Threads

- Usually a process has single thread of control
- In many situation it is desirable to have more than one thread in the same address space (inside a process).

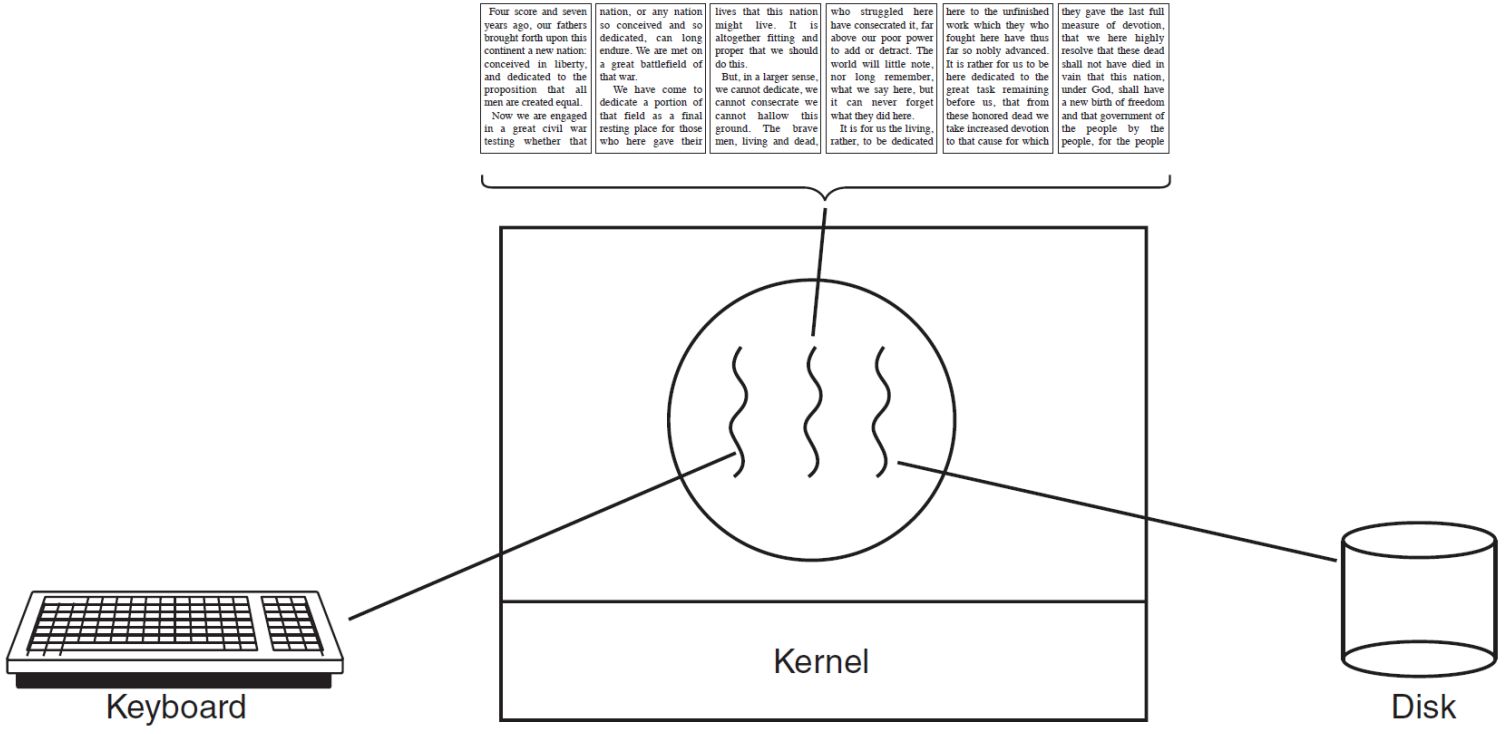
Thread Usage

Reasons for having multiple threads:

- Usually, multiple activities are going on at once in a process. Therefore one of them might block causing other activities to block as well.
- Threads are lighter weight than process. They are easier (faster; 10-100 times faster than creation of a process) to create and destroy
- Threads yield performance gain when threads are I/O and CPU bound.
- Real parallelism are possible in multi-processors systems, when using threads.

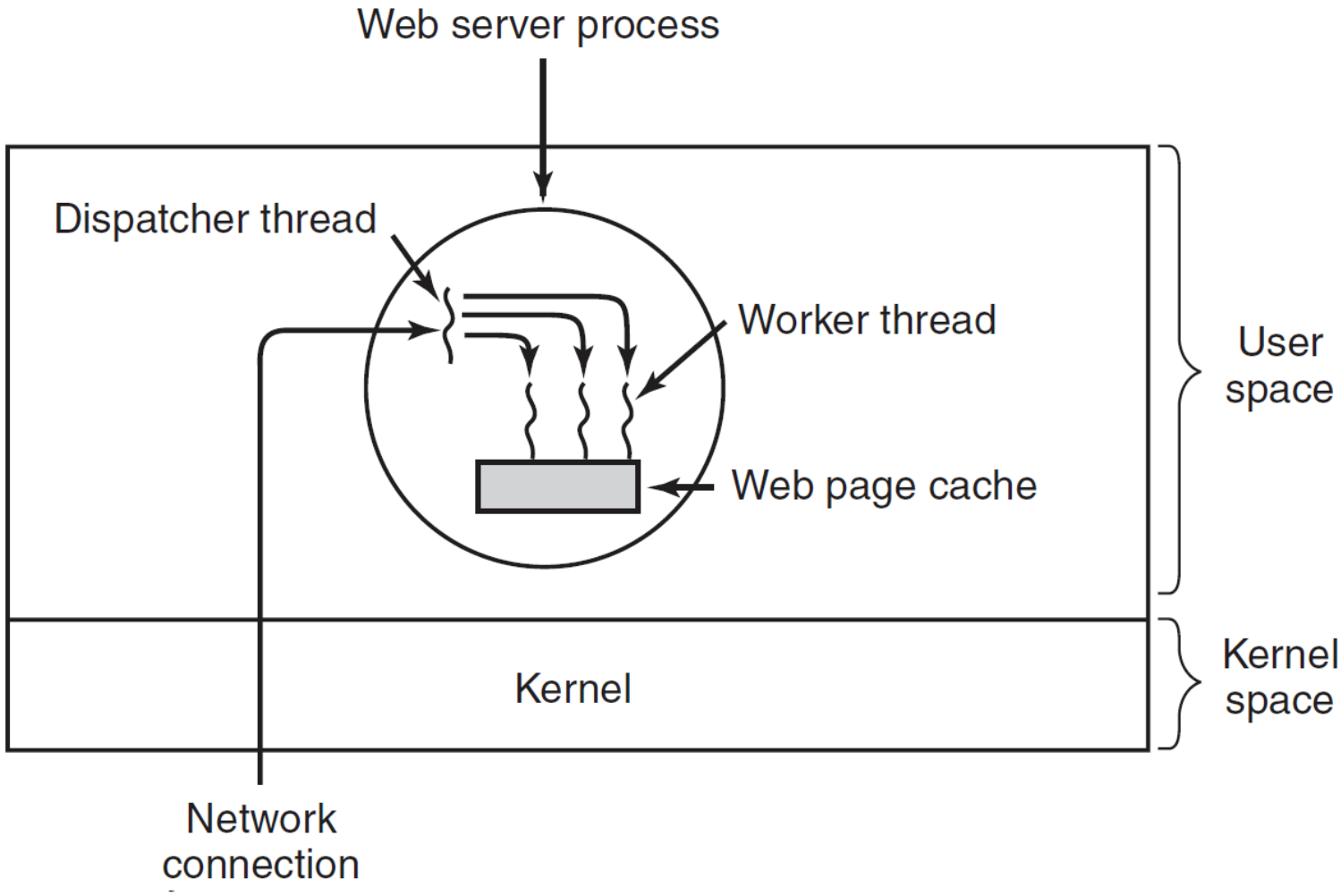
Threads

Thread Usage-word processor example



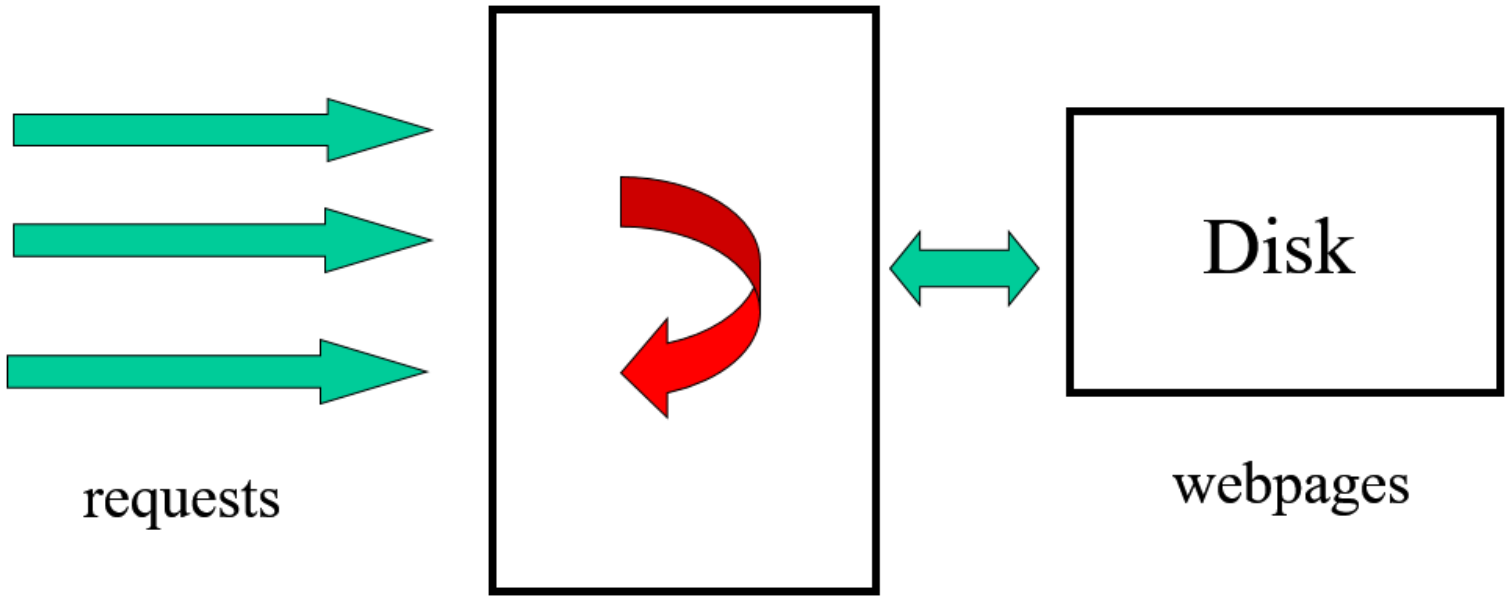
Threads

Thread Usage-Multithreaded web server



Threads

Thread Usage-Single threaded web server



Thread Usage-Multithreaded web server

- Dispatcher:

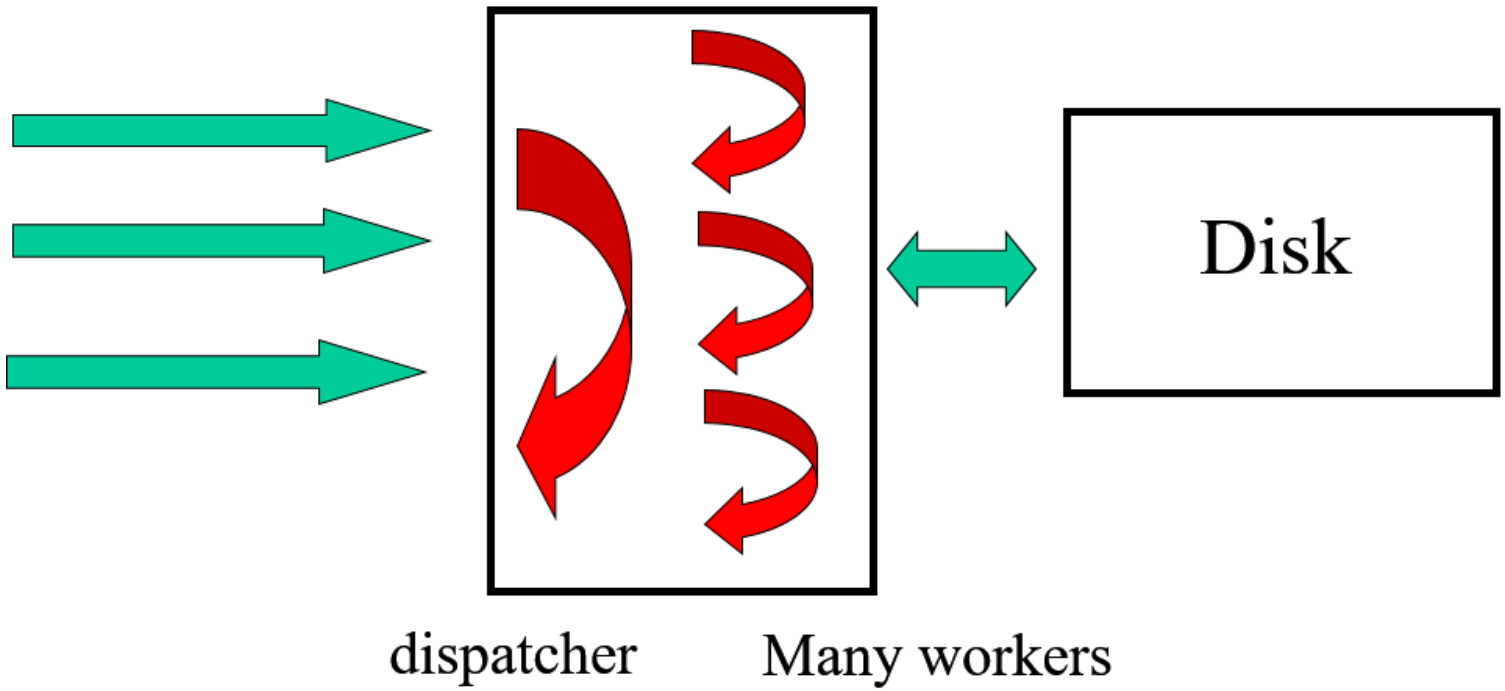
```
While (1) {
     get_request(&req);
     start_new_worker(req);
}
```

- Worker:

```
Worker_thread(req) {
    look_for_page_in_cache(req,&page);
    if(page_not_in_cache(&page))
         fetch_webpage(req,&page);
    return_page(req, page);
}
```

Threads

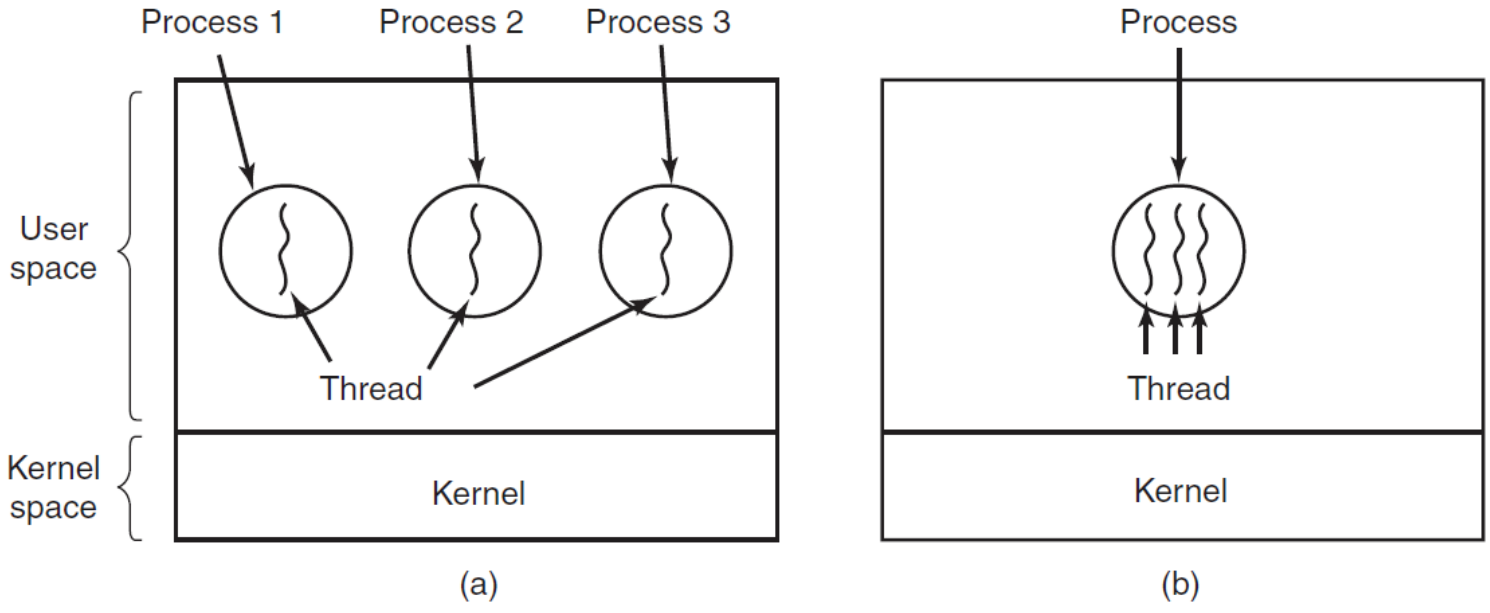
Thread Usage-Multithreaded web server



The Classical Thread Model

The process model is based on two independent concepts; resource grouping and execution. **The thread concept comes to separate them.**

Thread is a light weight process.



The three threads in three processes are doing different jobs, while the three threads in a process are doing related jobs.

Threads

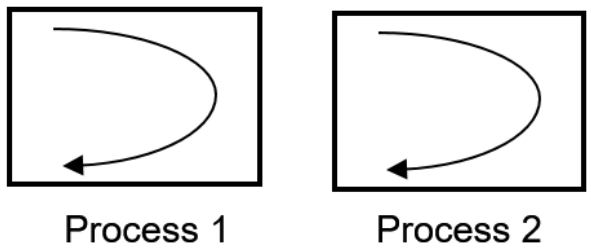
The Classical Thread Model

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

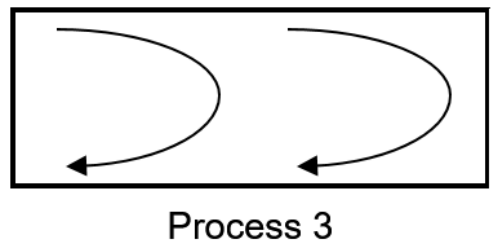
Shared between all threads in a process

private to a thread

Process vs. Threads



- Creating another process is expensive
- Switching between processes is expensive
- IPC is expensive



- Creating another thread is cheaper
- Switching between threads is cheaper
- IPC is cheaper

Process vs. Threads

Thread benefits

- Sharing address space and all of its data between threads in a process.
- Cheaper to create and destroy than a process.
- Performance. When there is a substantial computing and substantial I/O, having threads allows these activities to overlap, thus speeding up the application.
- Real parallelism is possible with threads on systems with multiple processors.

Threads

Will the following benefit from multiple threads?

- Multiplying huge matrices on a single processor.
- Multiplying huge matrices on multiple processors.
- UNIX shell

POSIX Threads

IEEE standard for portable threaded programs (1003.1c). It is called pthread and it is supported by many UNIX systems.

pthread_create

```
int pthread_create(
pthread_t *tid, /* thread id will be stored in tid      */
const pthread_attr_t *tattr, /* thread attribute object */
void*(*start_routine)(void *), /*routine name to be run */
void *arg);    /* argument passed to the routine        */
```

Example

```
start_servers( ) {
    pthread_t thread[nr_of_server_threads];
    int i;
    for (i=0; i<nr_of_server_threads; i++)
        pthread_create(&thread[i], //thread ID
        0,                         //default attributes
        server,                    //start routine
        argument);                 //argument
}

void *server(void *arg) {
    while(1) {
        /* get and handle request */
    }
}
```

Complications

```
func(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;

    pthread_create(&in_thread,
    0,
    incoming,
    r_in, l_out);  // Can't do this ...
    pthread_create(&out_thread,
    0,
    outgoing,
    l_in, r_out);  //Can't do this ...
    /* How do we wait till they're done? */
}
```

Multiple Arguments

```
typedef struct {
    int first, second;
} two_ints_t;

func(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};
    pthread_create(&in_thread,
    0,
    incoming,
    &in);
    ...
}
```

Threads

Some Threads Attributes:

- A thread may have local or global scope of contention.
  - That is, it may compete with all threads in the system for CPU time, or it may compete only with threads in the same task (process)
- A thread has a priority for scheduling.
  - Threads may use several scheduling methods, some of which use priority
- A thread may be detached.
  - Only non-detached threads may be joined
  - Join is to wait for another thread as wait to process

Thread attribute object

- The attributes of a thread are held in a thread attribute object, which is a **struct** defined in pthread.h.
- You can declare a pthread attribute in your code, but it can only be initialized or modified by the following functions:

```
int pthread_attr_init(pthread_attr_t *attr);
pthread_attr_setstackaddr();
pthread_attr_setstacksize();
pthread_attr_setdetachstate();
```

Thread attribute object

- Creating a thread using a **NULL** attribute argument has the same effect as using a default attribute:
  - Non-detached (joinable)
  - With a default stack and stack size
  - With the parent's priority
- To create threads with other attributes, the generated attribute object must be modified using the *pthread_attr_set* functions.

Thread attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
...
/* establish some attributes */
...
pthread_create(&thread, &thr_attr, startroutine, arg);
...
```

Notes

- Contrast this approach vs providing a long list of parameters.
- Release the storage created for the attributes by calling *pthread_attr_destroy*.

Thread stack size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);

...
pthread_create(&thread, &thr_attr, startroutine, arg);
...
```

Threads

What is wrong with this?

```
void func(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};

    pthread_create(&in_thread, 0, incoming, &in);
    pthread_create(&out_thread, 0, outgoing, &out);

    return;
}
```

Threads

What about this?

```
void func(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};

    pthread_create(&in_thread, 0, incoming, &in);
    pthread_create(&out_thread, 0, outgoing, &out);

    pthread_join(in_thread, 0);
    pthread_join(out_thread, 0);
}
```

Waiting for pthreads

- Use *pthread_join()* to wait for a thread to terminate.
- Prototype:

```
int pthread_join(thread_t tid, void **status);
```

- The *pthread_join()* function blocks the calling thread until the thread specified by *tid* terminates.
- The specified thread must be:
  - in the current process and
  - non-detached

Waiting for pthreads

- The exit status of the thread specified by *tid* is written to status when *pthread_join()* returns successfully.
- Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of **ESRCH**. (No item could be found that matches the specified value)

Thread termination

```
pthread_exit((void *) value);

return((void *) value);


pthread_join(thread, (void **) &value);
```

Thread termination

- An important special case arises when the initial thread, the one calling *main()*, returns from *main()* or calls *exit()*.
- This action causes the entire process to terminate, along with all its threads.
- So take care to ensure that the initial thread does not return from *main()* prematurely.
- Note that when the main thread merely calls *pthread_exit()*, it terminates only itself-the other threads in the process, as well as the process, continue to exist.
- The process terminates when all its threads terminate.

Detached threads

```
start_servers() {
    pthread_t thread;
    int i;
    for (i=0; i<nr_of_server_threads; i++) {
        pthread_create(&thread, 0, server, 0);
        pthread_detach(thread);
    }
    ...
}

server( ) {
    ...
}
```

Multiplying two matrices using threads (1)

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

#define M       3
#define N       4
#define P       5

int A[M][N];
int B[N][P];
int C[M][P];

void *matmult(void *);
```

Multiplying two matrices using threads (2)

```
main( ) {
    int i;
    pthread_t thr[M];
    int error;

    /*initialize the matrices ... */

    ...
    for (i=0; i<M; i++) { //create the worker threads
        if (error = pthread_create(&thr[i],0,matmult,(void *)i)) {
            fprintf(stderr,"pthread_create: %s", strerror(error));
            exit(1);
        }
    }
    for (i=0; i<M; i++)//wait for workers to finish their jobs
        pthread_join(thr[i], 0);
        /*print the results ... */
}
```

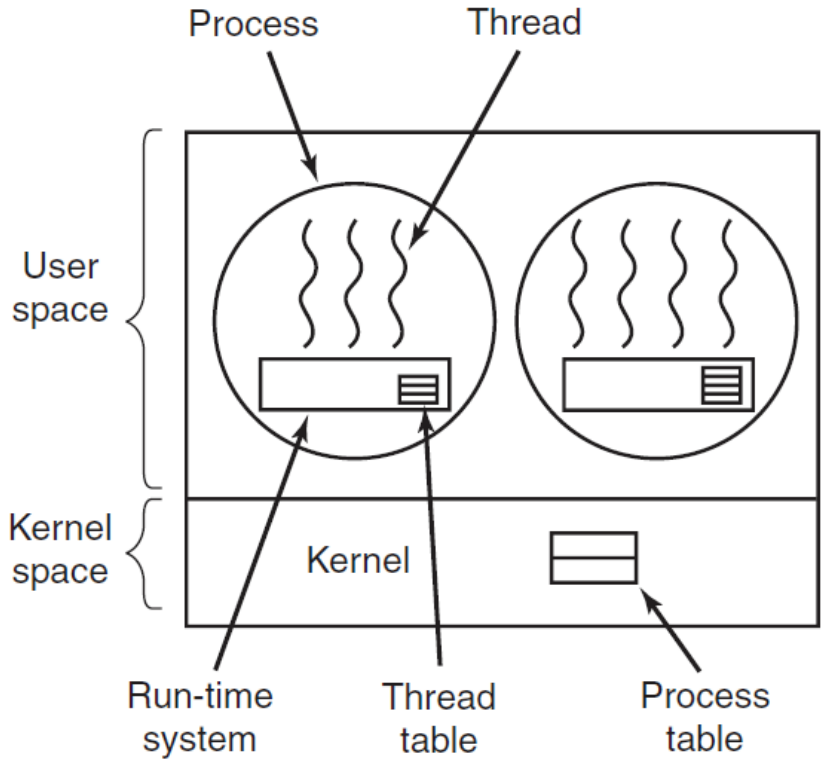
Multiplying two matrices using threads (3)

```
void *matmult(void *arg) {
    int row = (int)arg;
    int col;
    int i;
    int t;

    for (col=0; col < P; col++){
        t = 0;
        for (i=0; i<N; i++)
            t += A[row][i] * B[i][col];
        C[row][col] = t;
    }
    return(0);
}
```

Implementing Threads in User Space

A user-level threads package can be implemented on an operating system that does not support threads. All operating systems used to fall into this category, and even now some still do. With this approach, threads are implemented by a library.



Advantages of User Space Threads

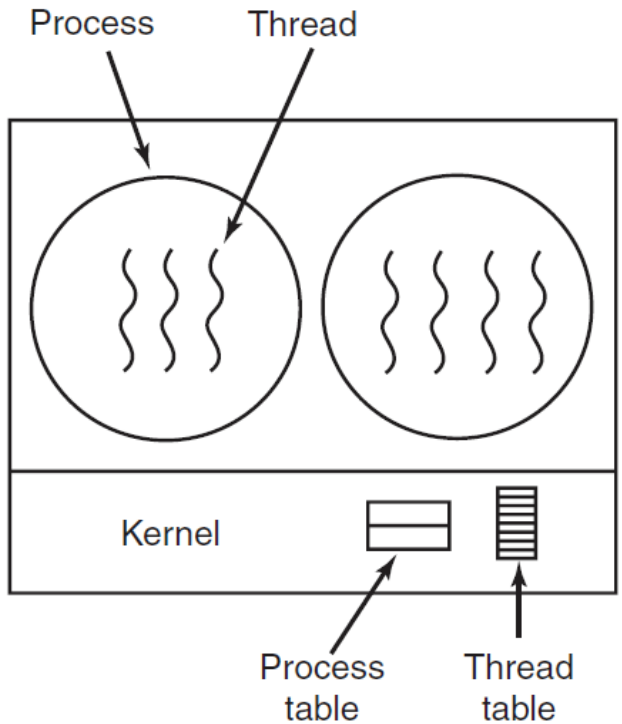
1. Switching between threads is faster in an order of magnitude (10 times) than trapping into the kernel.
2. User level threads allow each process to have its own scheduling algorithm.
3. Can be used in an OS that doesn't support threads.

Disadvantages

1. Blocking system call causes all threads in a process to be blocked.
2. A thread can run forever not allowing for other threads in a process to run. This is due to unavailable clock to make running thread stop execution.
3. Cannot take advantage of a multiprocessor

Implementing Threads in Kernel Space

No run-time system is needed in each process. Also, there is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.



Threads

Advantages of Implementing Threads in Kernel Space

1. Can take advantage of multiple processors
2. System call blocks only the thread which made the call

Disadvantage

Thread operations involve system calls (expensive)

Hybrid Implementation