

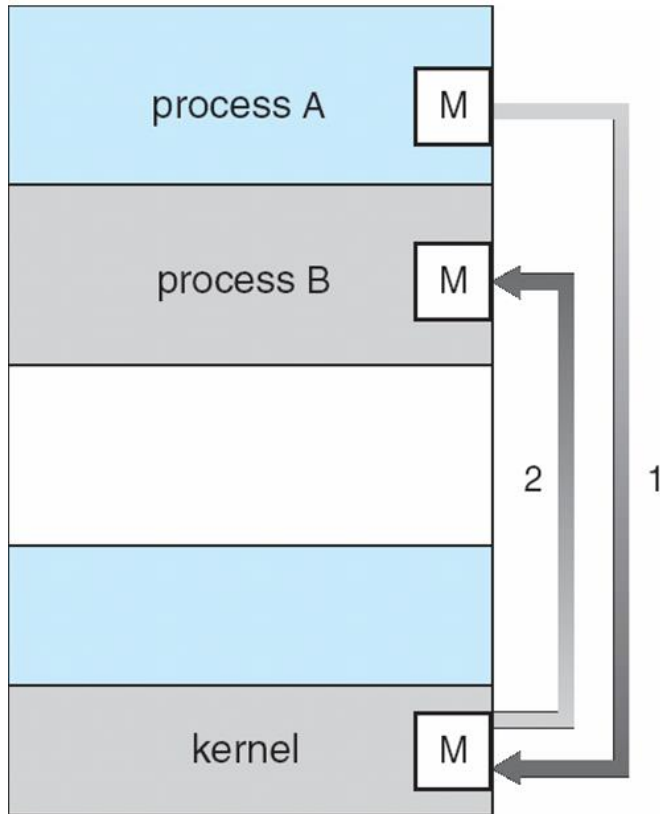
Process Management (II)

August 30, 2017

Interprocess Communication

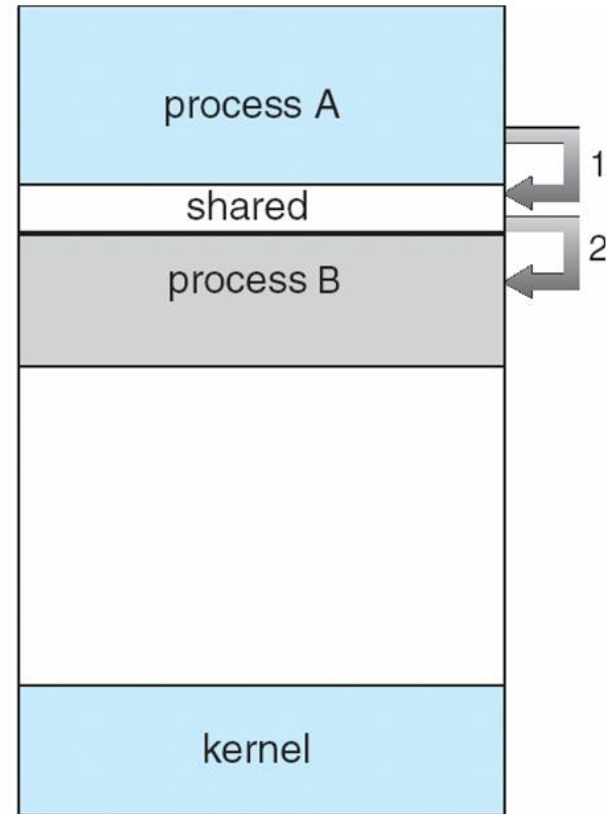
- ❏ Processes within a system may be **independent** or **cooperating**
- ❏ Cooperating process can affect or be affected by other processes, including sharing data
- ❏ Reasons for cooperating processes:
 - ❏ Information sharing
 - ❏ Computation speedup
 - ❏ Modularity/Convenience
- ❏ Cooperating processes need **inter-process communication (IPC)**

Communications Models



(a)

Message Passing

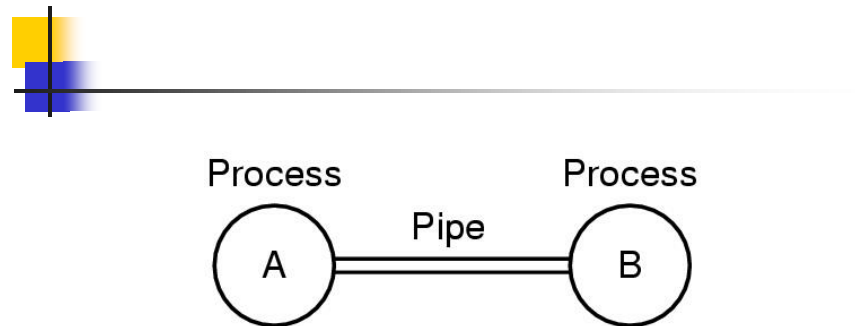


(b)

Shared Memory

Example of Message Passing: Unix Pipes

- ❏ Pipe sets up communication channel between two (related) processes.

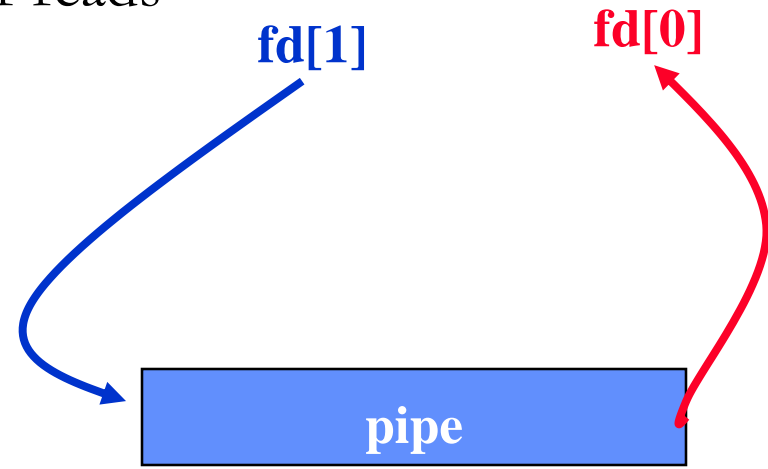


Two processes connected by a pipe

Unix Pipes

- ❏ One process writes to the pipe, the other reads from the pipe.
- ❏ Similar to reading from/writing to a file.
- ❏ System call:

```
int fd[2] ;  
pipe(fd) ;
```

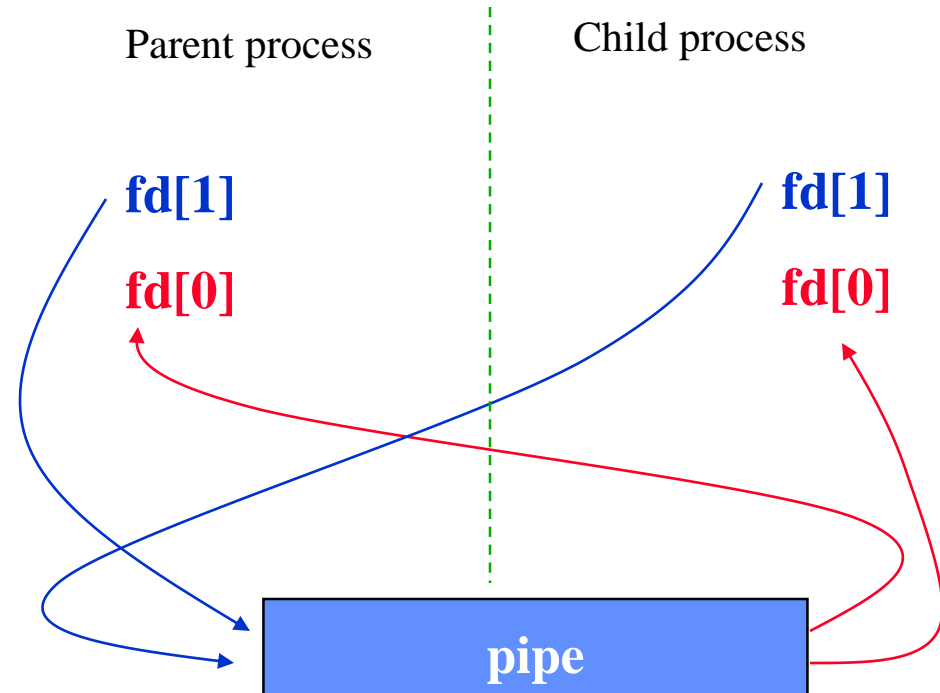


fd[0] now holds descriptor to read from pipe

fd[1] now holds descriptor to write into pipe

A Simple Example

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
char *message = "This is a message!!!";
main()
{ char buf[1024];
  int fd[2];
  pipe(fd); /*create pipe*/
  if (fork() != 0) { /* I am the parent */
    write(fd[1], message, strlen (message) + 1);
  }
  else { /*Child code */
    read(fd[0], buf, 1024);
    printf("Got this from parent!!: %s\n", buf);
  }
}
```



Synchronization

- ❏ Message passing may be either blocking or non-blocking
- ❏ **Blocking** is considered **synchronous**
 - ❏ **Blocking send** has the sender block until the message is received
 - ❏ **Blocking receive** has the receiver block until a message is available
- ❏ **Non-blocking** is considered **asynchronous**
 - ❏ **Non-blocking send** has the sender send the message and continue
 - ❏ **Non-blocking receive** has the receiver receive a valid message or null
- ❏ **Read** in pipe is blocking

Limitations of Pipes

- ❏ Processes using a pipe must come from a common ancestor
 - ❏ e.g., parent and child; siblings
- ❏ Pipes are not permanent
 - ❏ disappearing when the process terminates

Named Pipes (Called FIFO in UNIX)

- Similar to pipes, but with some advantages
 - A FIFO can be created separately from the processes that will use it
 - FIFOs look like files
 - Having an owner, size, access permissions
 - Permanent until deleted with `rm`

Creating a FIFO

- Using unix command line (calling system program):

```
$mkfifo <FIFO-NAME>
```

e.g., \$mkfifo fifo1

- Using C program

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int mkfifo(const char *pathname, mode_t mode);
```

- pathname: path-name of the FIFO

- mode: permission. It could be any combination of S_IRUSR (owner read), S_IWUSR (owner write), S_IRGRP (owner's group read), S_IWGRP (group write), S_IROTH (other users read), S_IWOTH (other write)

e.g., mkfifo("fifo1", S_IRUSR|S_IWUSR); //create FIFO fifo1 for owner's processes' communication


Open an Existing FIFO


```
#include <sys/types.h>
```


```
#include <sys/stat.h>
```


```
#include <fcntl.h>
```

```
int open( const char *pathname, mode_t mode );
```


 pathname: path-name of the FIFO

 open a FIFO to write: `open (FIFOName, O_WRONLY);`


 open a FIFO to read: `open (FIFOName, O_RDONLY);`

 After a FIFO has been open, read/write operations can be performed on it like with the pipe.

POSIX Shared Memory

 Process first creates shared memory segment

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR |  
    S_IWUSR); //a new shared memory segment is created  
segment_id = shmget(key, size,  
    S_IRUSR|S_IWUSR|IPC_CREAT); //if a SM segment  
    associated with key exists, its ID is returned;  
    otherwise, a new SM segment is created
```

 Process wanting access to that shared memory must attach to it

```
Shared_memory = (char *) shmat(id, NULL, 0);
```

 Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

 When done a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

A simple program

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main() {
    int segment_id;
    char *shared_memory;
    const int size = 4096;
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR);
    shared_memory=(char *)shmat(segment_id, NULL,0);
    sprintf(shared_memory, "Hi there!");
    shmdt(shared_memory);
    shmctl(segment_id,IPC_RMID,NULL); //remove the shared memory block
}
```

A simple program (cont.): Sharing a memory between parent and child processes

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main(){
    int segment_id;
    char *shared_memory;
    const int size = 4096;
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR);
    shared_memory=(char *)shmat(segment_id, NULL,0);
    if (fork() == 0){
        sprintf(shared_memory, "Hi, this is the child!");
        shmdt(shared_memory);
        exit(0);
    }else{
        wait(NULL);
        printf("%s\n", shared_memory);
        shmdt(shared_memory);
        shmctl(segment_id,IPC_RMID,NULL); //remove the shared memory
    } ...
}
```

Another example: Sharing a memory between two processes

```
#define KEY 9876
int main(){
    int segment_id;
    char *shared_memory;
    const int size = 4096;
    segment_id = shmget(KEY, size, S_IRUSR|S_IWUSR|IPC_CREAT);
    shared_memory=(char *)shmat(segment_id, NULL,0);
    sprintf(shared_memory, "Hi, this is the child!");
    shmdt(shared_memory); }
```

```
#define KEY 9876
int main(){
    int segment_id;
    char *shared_memory;
    const int size = 4096;
    segment_id = shmget(KEY, size, S_IRUSR|S_IWUSR|IPC_CREAT);
    shared_memory=(char *)shmat(segment_id, NULL,0);
    printf("%s\n", shared_memory);
    shmdt(shared_memory);}
```

Unix/Linux Signals

Signals are software interrupts

- Examples: A user hitting ctrl+c; a process sending a signal to kill another process.

In Linux, signals are defined at `/usr/include/bits/signum.h`, and every signal has a name that begins with characters SIG:

- SIGINT : when a user presses ctrl+c.
- SIGKILL
- SIGUSR1, SIGUSR2: you are free to define what they mean

Unix/Linux Signals

How to send a signal to a process?

- System program kill

example: `>kill -9 1000` //kill process with ID 1000

- System call kill

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

Unix/Linux Signals

How to handle a signal in a process?

- Define a handler function
- Make system call signal to link the function with a signal

Unix/Linux Signals

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sig_handler(int signo) {
    if(signo == SIGINT) printf("received SIGINT\n");
}

int main(void) {
    if(signal(SIGINT, sig_handler) == SIG_ERR)
        printf("Cannot catch SIGINT\n");
    while(1) sleep(1);
}
```

Quiz

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main( ) {
    int i;
    for(i=0;i<2;i++) fork();
}
```

How many processes are created while the above program is executed?

Thread (I)

Kernel Threads

September 1, 2017

Example of Multi-Process Collaboration

 Finding the lines containing key word “Linux” in all .txt files in the current directory

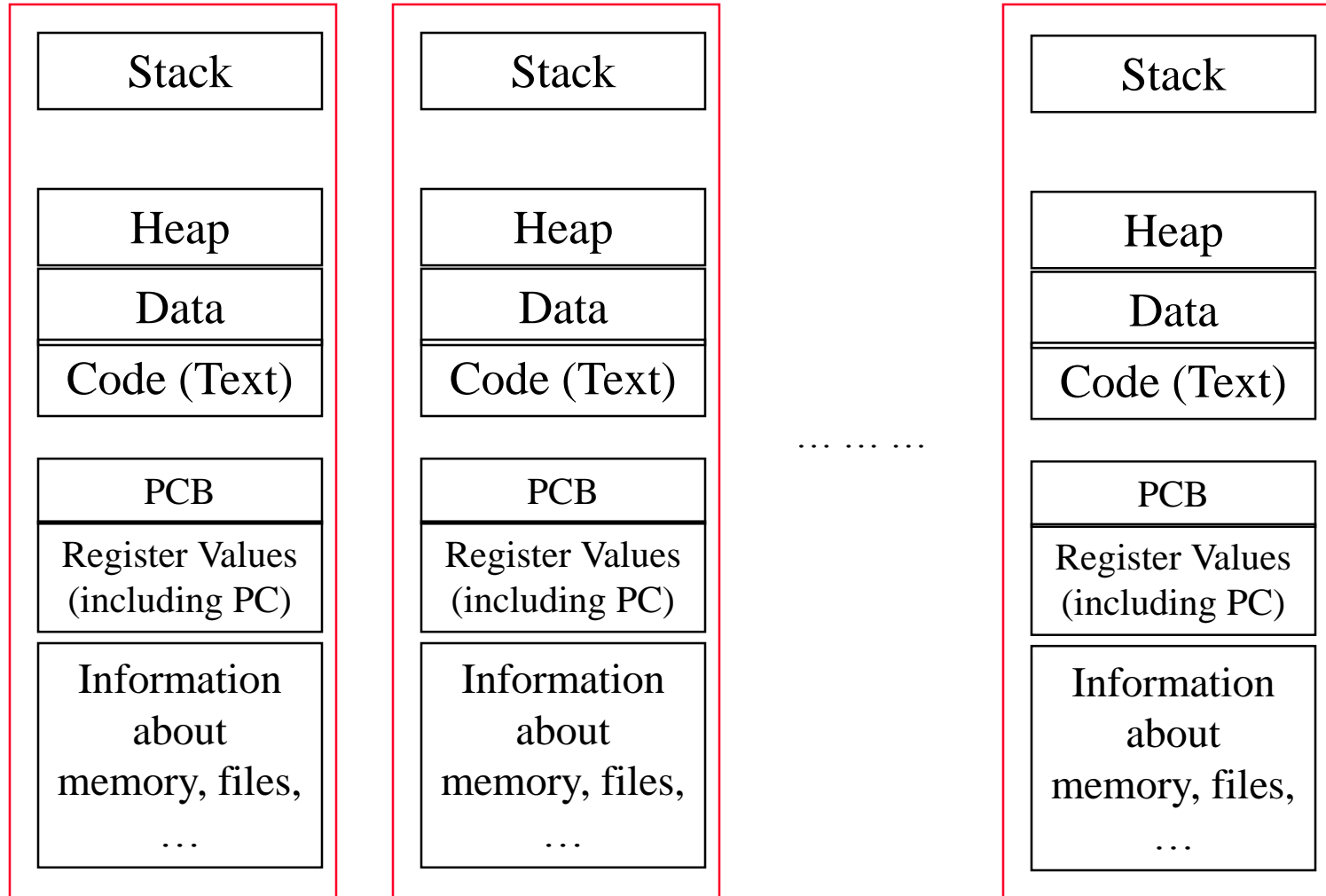
 Linux command line:

```
grep “Linux” *.txt
```

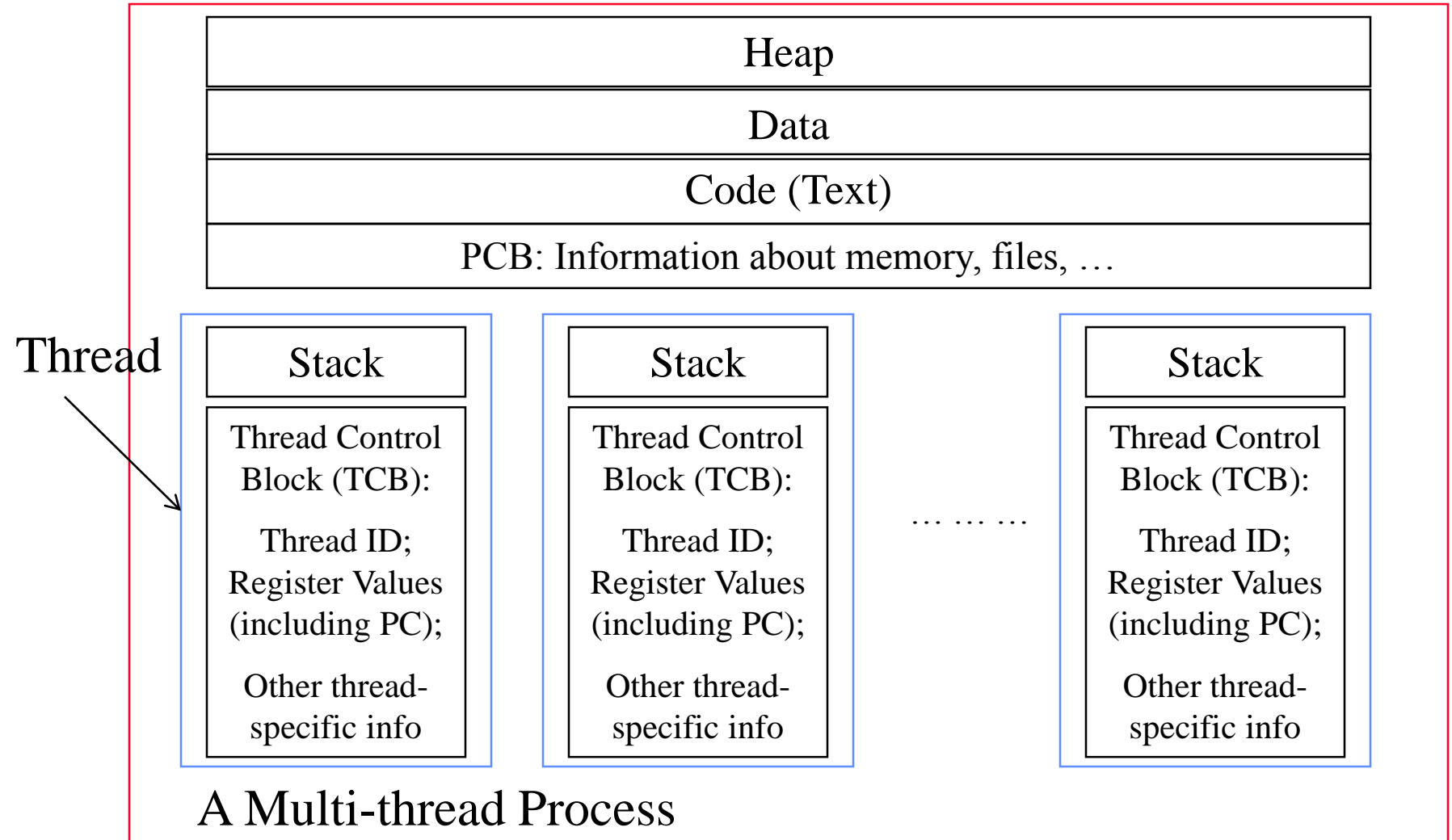
 Using multi-process collaboration to do the job?

```
...
int numFiles; //total number of files
char fileNames[MAX_NUM_FILES][255]; //array storing the file names
int i;
for(i=0; i<numFiles; i++)
    if (fork() == 0){
        FILE *fp = fopen(fileNames[i], “r”);
        ... //search for lines containing “Linux” in the file
        exit(0);
    }
```

Multi-Process





A Process with Multiple Threads



Benefits

Responsiveness (similar to multi-process)

-  Allowing a program to continue running even if part of it is blocked
-  Allowing a program to perform multiple tasks concurrently


Resource sharing

-  Threads of the same process share the memory and resources

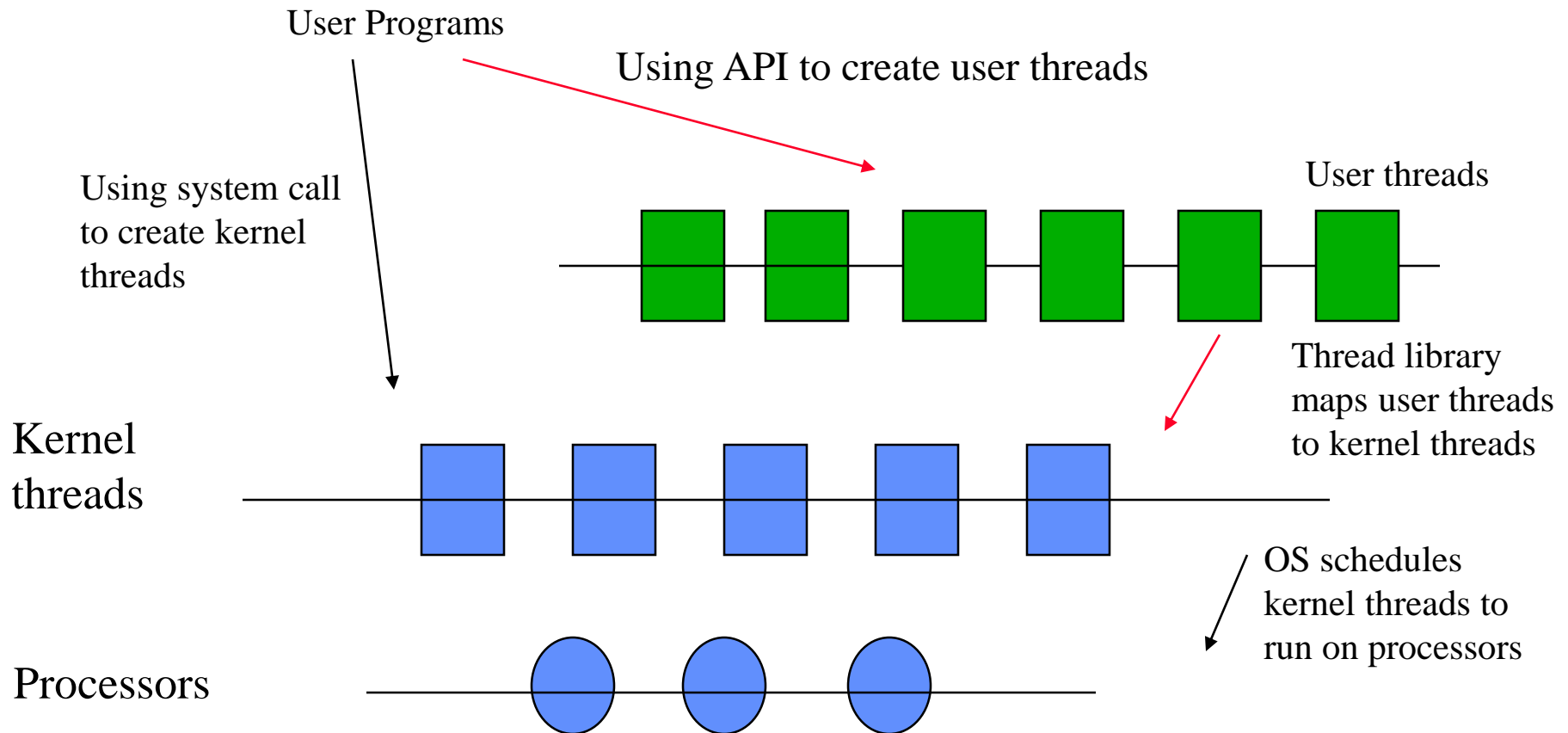
Economy

-  Allocating memory and resources for process creation is costly.

Scalability (similar to multi-process)

-  A single-thread process can only run on one processor regardless how many are available; multi-threaded process can increase parallelism on a multi-processor machine.

Threads in a Computer System



Kernel Threads

- ❏ Directly created/managed/scheduled by the OS kernel
- ❏ Virtually all contemporary OSes support kernel threads

Linux Kernel Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call

```
#define _GNU_SOURCE
```

```
#include <sched.h>
```

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

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.

Example

```
#include _GNU_SOURCE
```

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

```
#include <sched.h>
```

```
#include <unistd.h>
```

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

```
#include <fcntl.h>
```

```
int variable, fd;
```

```
int do_something(); //function to be executed by the new task (thread)
```

Example

```
int main() {  
    void *child_stack;  
    char tempch;  
    variable = 9;  
    fd = open("test.file", O_RDONLY);  
    child_stack = (void *)malloc(16384); child_stack += 16383;  
    printf("The variable was %d\n", variable);  
    clone(do_something, child_stack, CLONE_VM|CLONE_FILES, NULL); /*A*/  
    sleep(1);  
    printf("The variable is now %d\n", variable);  
    if (read(fd, &tempch, 1) < 1) {  
        perror("File Read Error"); exit(1);}  
        printf("Parent could read from the file\n");}  
}
```

Example

```
int do_something() {  
    char tempch;  
    variable = 42;  
    if (read(fd, &tempch, 1) < 1) {  
        perror("File Read Error");  
        _exit(1);  
    }  
    printf("Child could read from the file\n");  
    close(fd);  
    _exit(0);  
}
```

Example

1. What is the output of the program?

2. What if Line A is changed to:

```
clone(do_something, child_stack, CLONE_VM, NULL);
```

or

```
clone(do_something, child_stack, CLONE_FILES, NULL);
```

or

```
Clone(do_something, child_stack, 0, NULL);
```