Threads

CS 342 – Operating Systems

Ibrahim Korpeoglu
Bilkent University
Computer Engineering Department

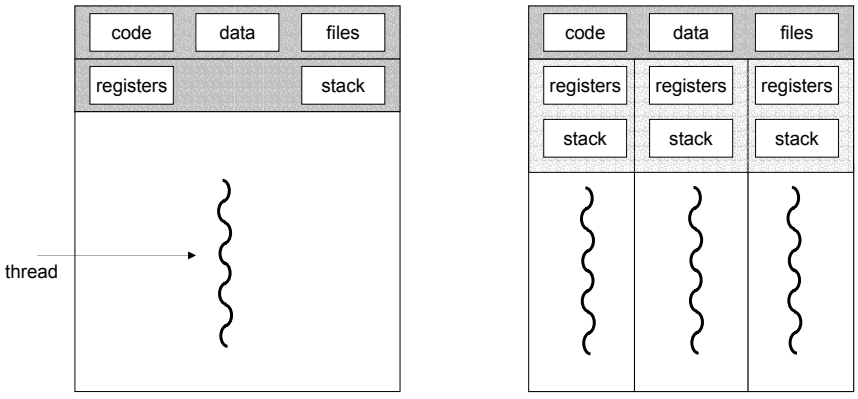
- A process has an address space and a single thread of control:
  - Single thread of control means that there is one PC register value that points to a single location in the program code and that is updated with every instruction execution.
- There is sometimes need for multiple threads of control line the same address space running in quasi-parallel.

- Looking to the process model
  - A process has two functions:
    - 1) way of grouping together a related set of resource to achieve a task.
      - Resource include: core image (text, data), open files, child processes, pending alarms, signal handlers, accounting information, etc.
    - 2) a process has a thread of execution (sequence of instructions)
      - Includes a program counters.
      - Registers
      - Stack: execution history of procedures

Thread concept:

- Adds multiple execution sequences (threads) to the same process environment so that the threads are using (sharing) resources together.
- Threads run in parallel.
- They are also called light-weight processes
  - Leight-weight, since it is much less costly to create a thread and to context-switch between threads.

Single versus multi-threaded processes



Single-thread process

Multi-threaded process

- All threads have
  - The same address space
    - They share global variables
  - Same set of open files
- All threads have their
  - Own PC register value
  - Own stack and stack pointers
    - Local variables of functions can not be shared
  - Set of CPU registers

Items shared and not shared

- Per process items (shared)
  - Address space
  - Global variables
  - Open files
  - Child processes
  - Pending alarms
  - Signals and signal handlers
  - Accounting information
- Per thread items | (not shared)
  - Program counter
  - Registers
  - Stack
  - State

Threads

- Like a process, a thread can be in one of 3 states: ready, blocked (waiting), running
- A thread may wait:
  - For some external event to occur (such a I/O completion)
  - For some other thread to unblock it.
- When a program runs, its started as a single-threaded process
  - Then it can create other threads by calling functions such as thread_create().

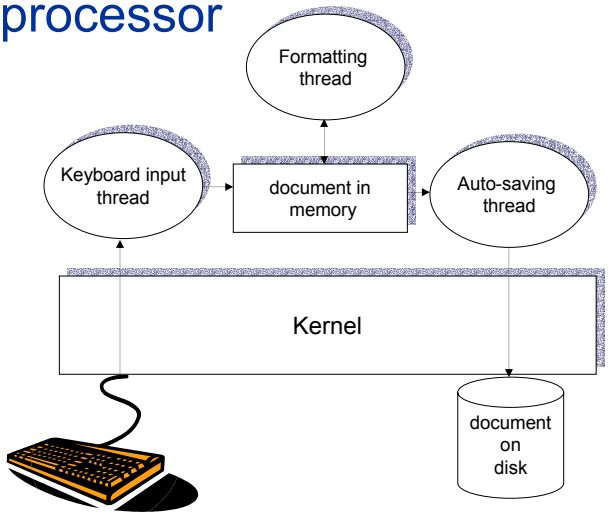
Thread functions

- thread_create()
  - Used to create a thread from an other thread (process)
  - One argument should be the name of function new thread will execute when thread is created.
- thread_exit()
  - When a thread finishes its task, it calls this function and it is no more schedulable.
- thread_wait()
  - By calling this function one thread can wait an other thread to exit. Calling thread is blocked.
- thread_yield()
  - By calling this function, a running thread may voluntarily relinquish CPU so that some other thread can run.

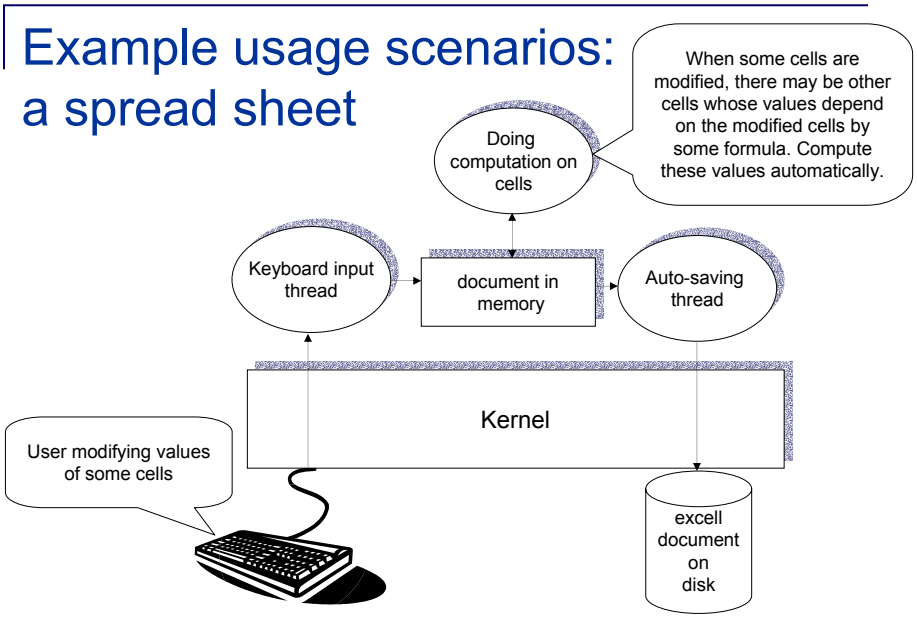
When we may need to use threads

- 1) In many applications, many activities may be going on at once.
  - By use of threads, we can parallelize these activities
  - The programming model can become simpler sometimes if we use threads.
- 2) Creation of threads are much faster than creation of processes (since they don't have resources attached to them).
  - May be 100 times faster
- 3) When there is substantial I/O and CPO cound operations in an applications and if these operations can be overlapped, then use of threads will increase the performance of the application (speed up).

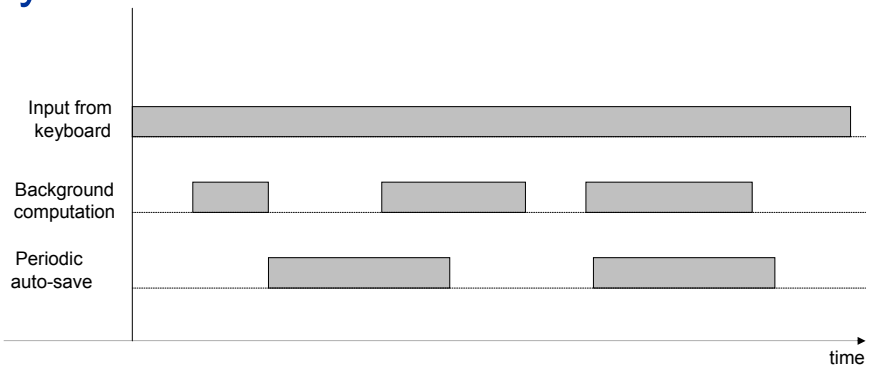
Example usage scenarios: a word processor



Example usage scenarios: a spread sheet



Concurrent and cooperative activities by different threads

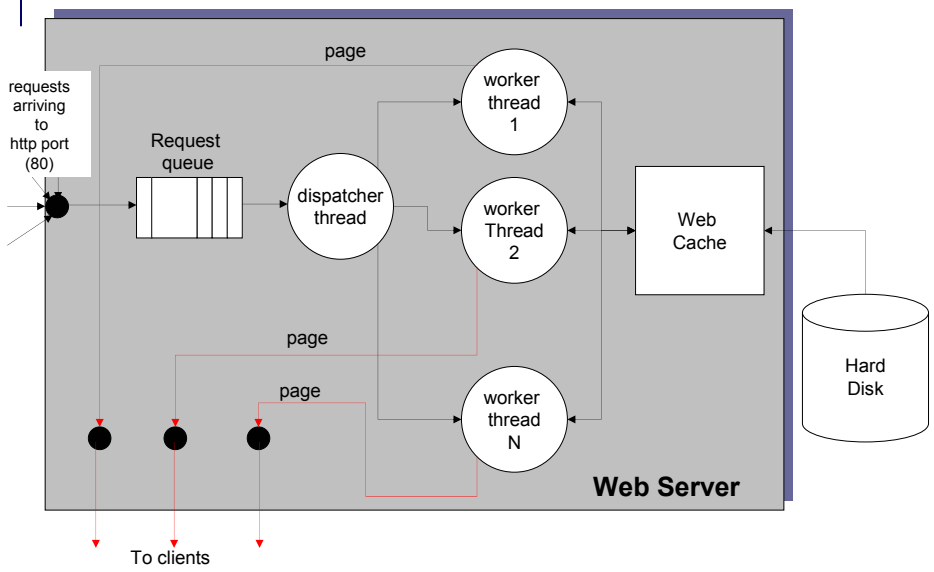


Example usage scenarios: web server

- A web servers
  - Receives web page requests from clients using HTTP protocol
  - Retrieves these pages from hard-disk into memory and then sends them to the appropriate client.
- A web server may have a cache:
  - Frequently requested pages can be kept in memory (in part of memory, which is called web cache)
    - A request file is served from cache (very fast) if it exists in the cache
    - A requested file is served from hard-disk if it does not exists in the cache (much slower)

Example usage scenarios: web server

- Multiple threads can be used in the application to better (in a fast manner) serve the requests.
  - A dispatcher thread
    - Created at program startup (main thread)
  - One or more threads to serve from cache or disk (worker threads)
    - Created dynamically.
    - There might be a limit in the number of worker threads.
      - If limit is reached, the requests need to be queued at the dispatcher.



If we did not use threads

- Web serve application performance (speed) will be degraded
  - If a page is not in cache, web server would retrieve the page from disk
    - This will take quite a while.
    - During this time, CPU will be idle or will not be used by web server process since web server is blocked waiting for disk I/O to complete.
- There is an other programming approach to solve this:
    - Use of non-blocking I/O

Non-blocking I/O

- By default I/O operations are blocking:
  - Example: a read() system call will block the process until it finishes successfully or unsuccessfully.
- These system calls or library functions can be done non-blocking by setting a special flag.
- In non-blocking operation, process calls the system call, but if there is no immediate data available for example to read, the call immediately returns.
  - The process then can continue doing some other task
  - The process can again call read() later to check if data arrived.

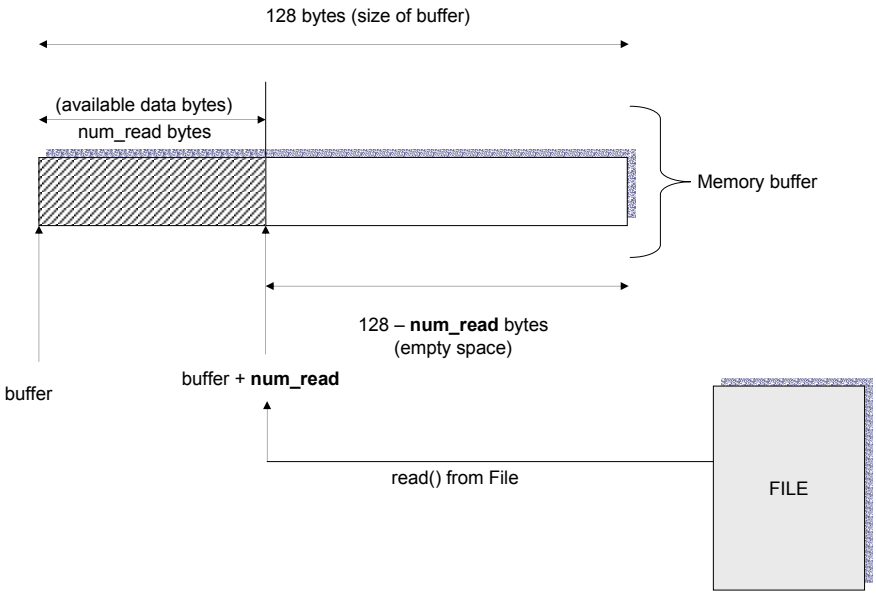
Non-blocking I/O

We can use fcntl() (file control) function to set some flags about the file that is opened.

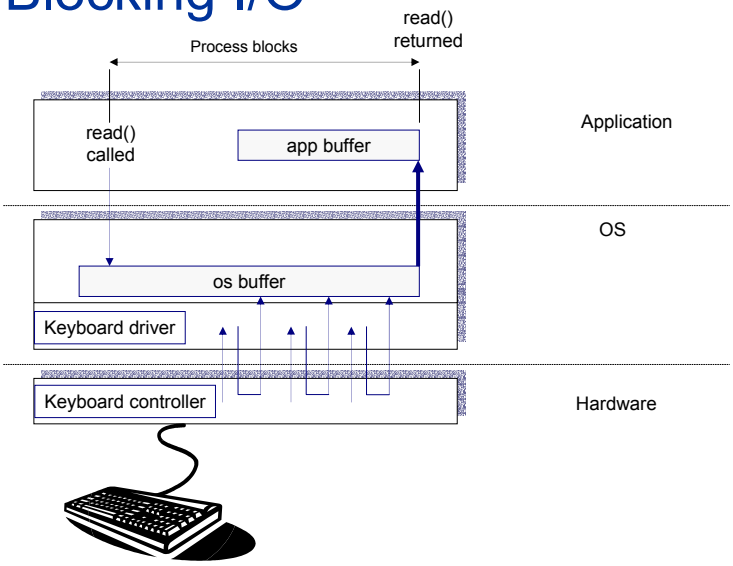
one of these flags sets if the operations on the file will blocked or non-blocked. It is set in the following way in Unix.

```
int   flags;
int   fd; /* file descripter */
void
main() {
      fd = open("myfile.txt", R_ONLY);

      if ((flags = fcntl(fd, F_GETFL, 0)) < 0)  /* first get file flags */
            printf("F_GETFL error"),

      flags = flags | O_NONBLOCK;   /* now set the corresponding bit for
      Non-                                  blocking I/O)*/
      if (fcntl(fd, F_SETFL, flags) < 0)   /* set new flags */
            printf("F_SETFL error");
}
```

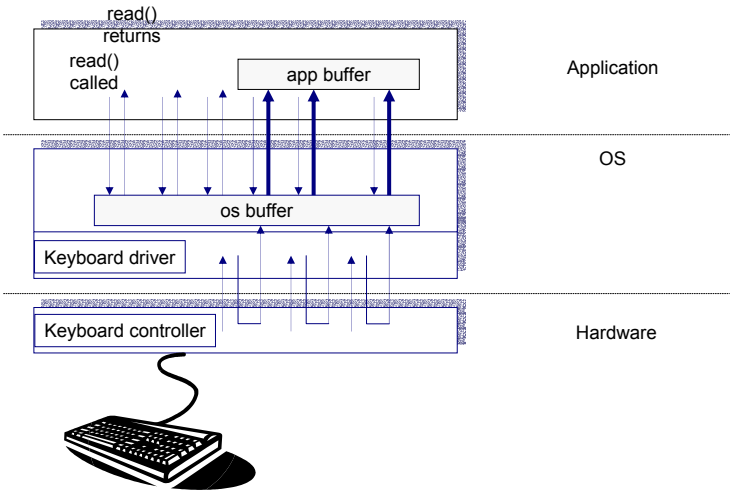
```
#include <sys/error.h>
int fd;   char buffer[4096];  int n;  int num_read = 0;
Void main() {
        fd = fileno(stdin); /* we want to read from keyboard */
        /* assume we have set fd so that it is non-blocking – like previous slide */
        while ( 1 ) {
                n = read(fd, (buffer + num_read), (128 - num_read));

                if (n < 0) {
                        if (errno != E_WOULDBLOCK) {
                                printf("there is some fatal error!\n");
                `               exit(1);
                        }  /* else go the start of while loop */
                }
                else if (n == 0) {
                        printf("we have reached end-of-file\n");
                        break;
                }
                else {    /* we managed to read some bytes */
                        num_read += n;
                        if (num_read >= 128)
                                break;
                };
                /* do some other work here if you want */
        }
        printf(" we have read 128 bytes from keyboard using non-blocking I/O");
}
```

Blocking I/O



Non-blocking I/O



Non-blocking

- Non-blocking I/O do not fit sequential programming model.
  - It starts an other operation before read completes.
  - It is harder to program
  - Threads are easier to program logically.
  - Blocking system calls makes the programming easier.
    - But we pay performance penalty.
- Finite state machines are used in non-blocking I/O to save the states of computations.

Server construction methods

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite state machine	Parallelism, non-blocking system calls, interrupts.

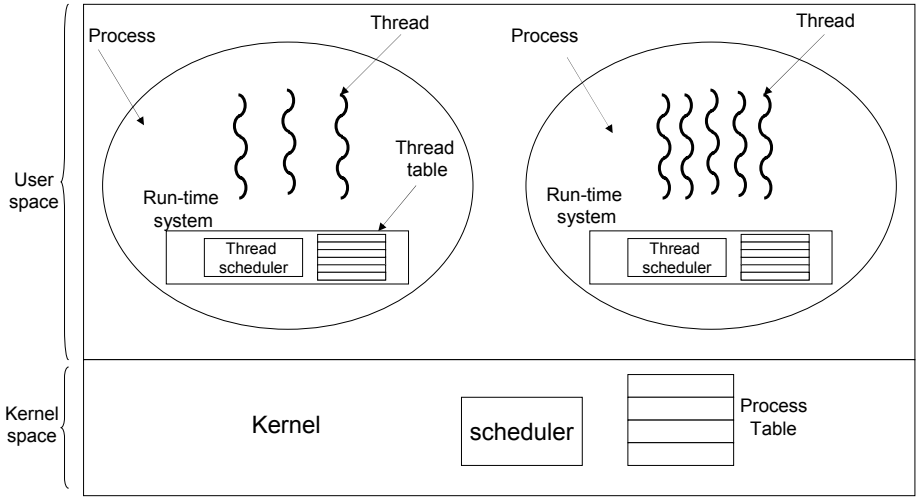
Implementing Threads

- There are many ways to implement threds:
  - In user space (as a library)
  - In kernel
- Each have advantages and disadvantages.

Implementing Threads in User Space

- Put the thread package entirely in user space.
- The kernel knows nothing about threads.
  - For old kernels that do not support threads
- Kernel is managing single-threaded processes
- All user level implementations have the same general structure.

Implementing Threads in User Space



- Run-time system: collection of procedures that manages threads.
  - thread_create, thread_exit, thread_wait, thread_yield.
  - Manages the thread table
- We have a separate thread table per each process
  - Each entry keeps info about one thread:
    - PC register, SP register, other registers, state, etc.

Switching from one thread to an other

- When a thread want to block locally (like waiting for an other thread) it calls a run-time system procedure.
- The procedure checks if the thread must be put into blocked state:
- If so, the thread is put into blocked state
  - The state info of the thread is stored in the thread table
  - An other thread is scheduled to run.
  - The state info of the other thread is loaded into CPU.
- This switching of threads can be implemented in user space very fast, since
  - we do not trap into kernel,
  - We don't need process context switch.
  - We just call local procedures.
  - No memory cache need to be flushed.
- Each process can also have its own customized thread scheduling algorithm.

Handling system calls

- Implementation of blocking system calls is tricky.
- When a thread blocks on a system call (like read), the process, hence all other threads will block too.
  - This is not desirable, since we want some other thread running if one blocks on a system call.
- Two ways to solve problem:
  - Modify system call so that they are non-blocking
  - Re-implement procedures in the I/O library that correspond to the system calls

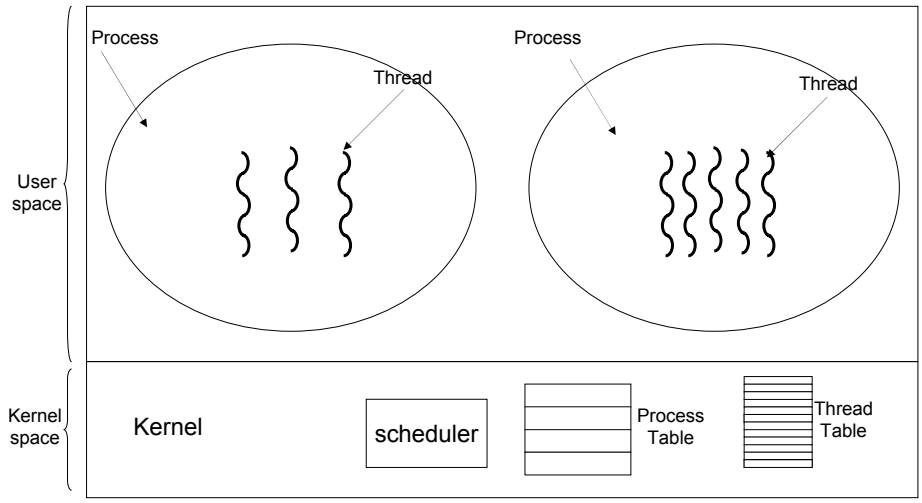
Handling system calls

- 1) use of non-blocking I/O
  - Requires changes of system calls semantics and therefore requires modifications to OS.
    - Not desirable
- 2) Modifying I/O library
  - Some OSs allow the application check if a system call would block before calling the system call.
  - Unix, for example, have the select system calls, that can be non-blocking and can check if a read() (or write()) operation from a file descriptor would block.
  - Using this select, we can rewrite the read() library function, so that it uses select() before calling the actual read system call.
    - If read system call would block, we don't call it. (an other thread is run)
    - If read system call would not block, we just call it and retrieve data.

Thread Scheduling for user space threads

- A thread should voluntarily stop, otherwise, it will run for the whole process time slice and no other thread will be able to run
  - We don't have clock interrupt for threads.
- The run-time system can request clock interrupt for every 1 second, for example, but this is also messy to program and may not be very efficient also.

Implementing Threads in Kernel Space



Implementing Threads in Kernel Space

- Kernel keeps track of threads.
- Kernel has a thread table.
  - Kernel also has a separate process table
- An entry per thread is kept in the table
  - That entry contains info that is a subset of process table entry info
  - Info includes: the PC register value, Sp value. Values of some other registers, state of thread, etc.

Implementing Threads in Kernel Space

- - A blocking call from a thread to wait for an other thread causes a trap to OS.
  - Therefore, these system calls are most costly compared to their user level thread implementation
- + When a thread blocks:
  - OS can run an other thread from the same process
  - OS can run a thread from some other process
  - OS can run an other process.
- - Thread creation and termination is more costly compared to user level thread implementation since:
  - All thread operations causes trap into OS: change from user mode to kernel mode, etc.
- + Kernel thread do not require any new non-blocking system calls or modifications to existing system calls or I/O library.

Pop-Up Threads

- Threads that are creating with every incoming network service request message.
  - No state associated in with the thread in the beginning of its execution: therefore, we do not have to restore the state for the pop-up thread
  - This enables fast initiation of processing of the incoming message
- The newly created serves the incoming message and terminates when service finished.