Chapter 5.2

Process Management

Thread Model. Process Scheduling

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

April 08, 2020

5.2.1

Purpose and Contents

The purpose of this subchapter

- Defines threads and the way they can be used
- Presents scheduling principles and algorithms

5.2.2

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, pg. 71 – 100, pg. 132 – 151

5.2.3

Contents

5.2.4

1 Threads

1.1 Definition of Concepts

Thread Definition

- describes a **sequential**, **independent execution** within a process
  - execution = the memory path followed in the code by the IP register
- **there could be more**
  - simultaneous and independent executions in the same process
  - $\Rightarrow$ **threads in a process**

5.2.5

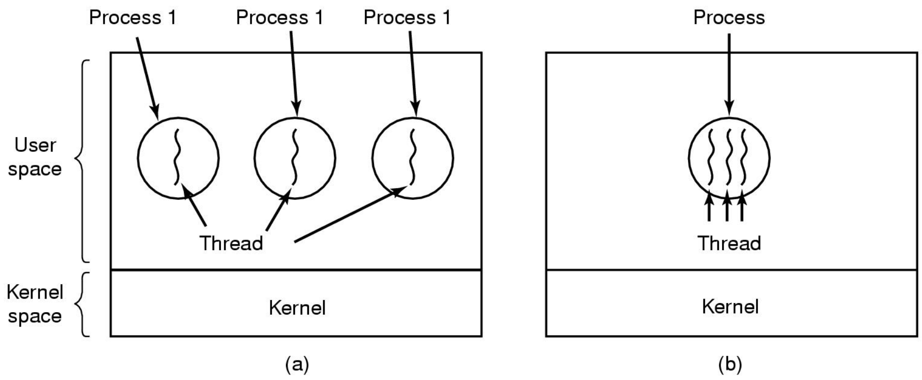


Figure 1: Taken from Tanenbaum

Modeling The Execution

- **process** model
  - models "**an entire computer**" used for executing some user application
  - composed by both
    - process' resources
    - process' execution
  - **isolates resources** of one process by that of others
- **thread** model
  - models **the processor(s)** given to a process
  - makes distinction between the two components of a process
    - the process describes the resources
    - the thread describes the execution
  - if multiple independent executions could be identified in a process
    - there could be more threads of that process
    - one thread $\leftrightarrow$ one execution
  - threads of a process **share resources** of that process
  - threads of a process (normally) **not visible to other processes**

5.2.6

Multiprogramming and Multithreading

- multiprogramming = multiple processes managed simultaneously
- multithreading = multiple threads in the same process managed simultaneously

5.2.7

Thread's Components. Description

- **threads** of the same process
  - **share all the resources of** that **process**: memory, open files etc.
  - $\Rightarrow$ anything (i.e. inside the process) is accessible to all threads (of that process)
- each thread is an independent execution
  - $\Rightarrow$ **each thread has also its own resources**
  - that describe the corresponding independent execution
- thread's specific resources
  - **machine registers** (e.g. the IP register)
  - **the stack**

5.2.8

Thread's Components. Illustration

5.2.9

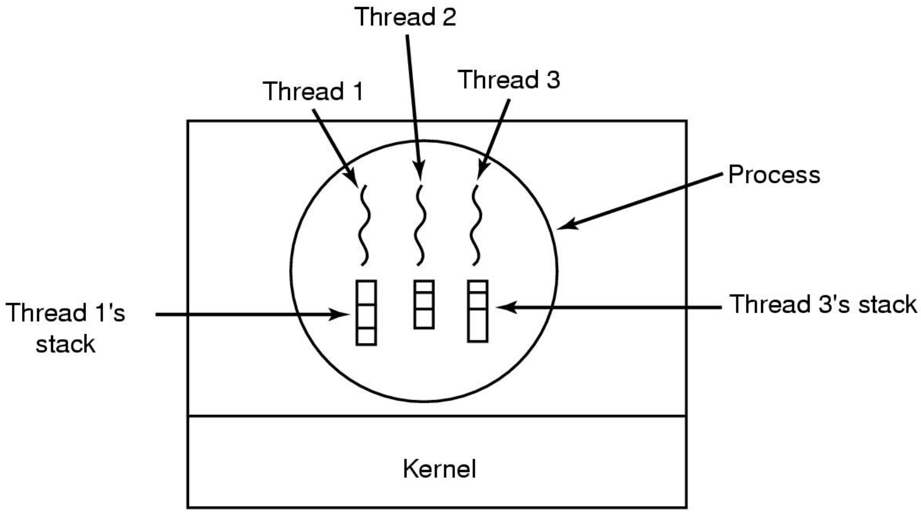


Figure 2: Taken from Tanenbaum

Threads Relationship

Paraphrase a rule from "Animal Farm" by George Orwell 5.2.10

Threads Relationship (cont.)

All threads are equal, but the "main" thread is "*more equal than the others!*" 5.2.11

Consequences of "All threads are equal"

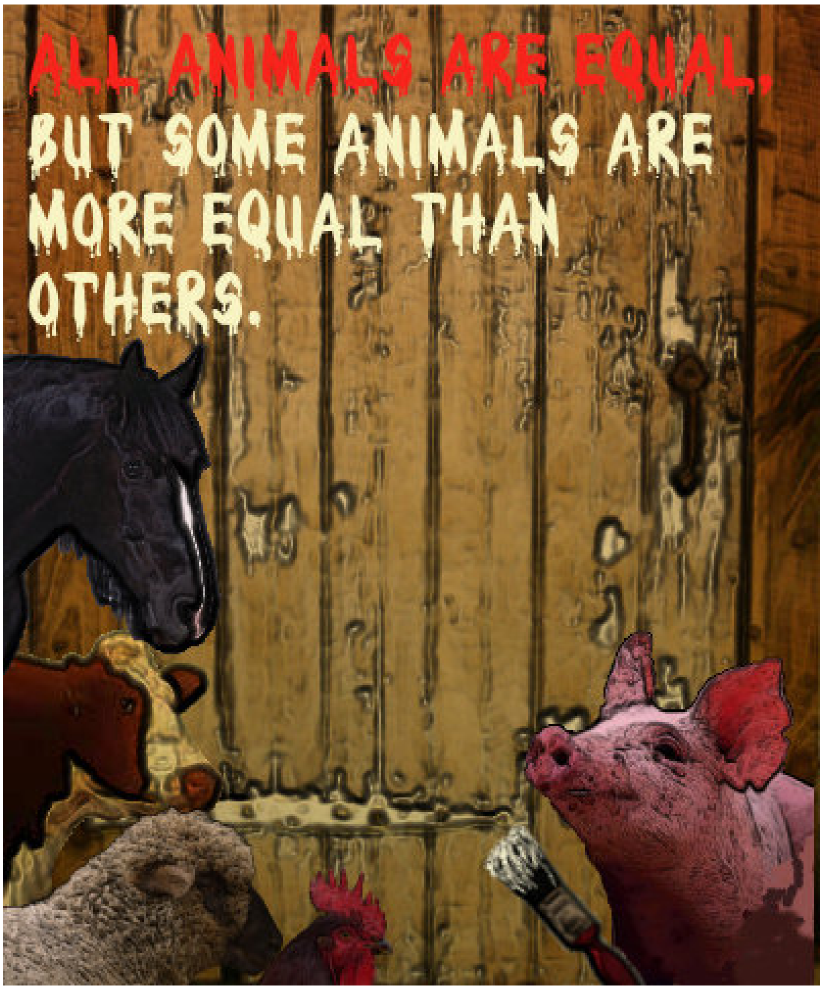
- **there is no protection between threads of the same process**
  - there is no need, actually
  - created by the application itself to execute code inside the application
- **there is no restriction on them regarding their scheduling**
  - any one could be scheduled any time, depending on the system's scheduling policy
  - we have no control on thread scheduling
  - if needed, **we must synchronize the threads**

5.2.12

Consequences of "All threads are equal" (cont.)

- **. . . but the main one is "*more equal than the others*"**
  - it is the first one created: **executes the *main()* function**
  - it is the ancestor of all other threads, i.e. initiates creation of the other threads
  - if it terminates returning from the "`main()`" function
    - the entire process (and all threads) terminates
    - ⇒ it make sense for it to **wait for the termination of all other threads**
  - **though**, if it terminates using the thread termination syscall, the other threads survive
    - see difference between `exit()` and `pthread_exit()` in Linux
    - see a discussion about this at https://devblogs.microsoft.com/oldnewthing/20100827-00/?p=13023

5.2.13


ALL ANIMALS ARE EQUAL,
BUT SOME ANIMALS ARE MORE EQUAL THAN OTHERS.

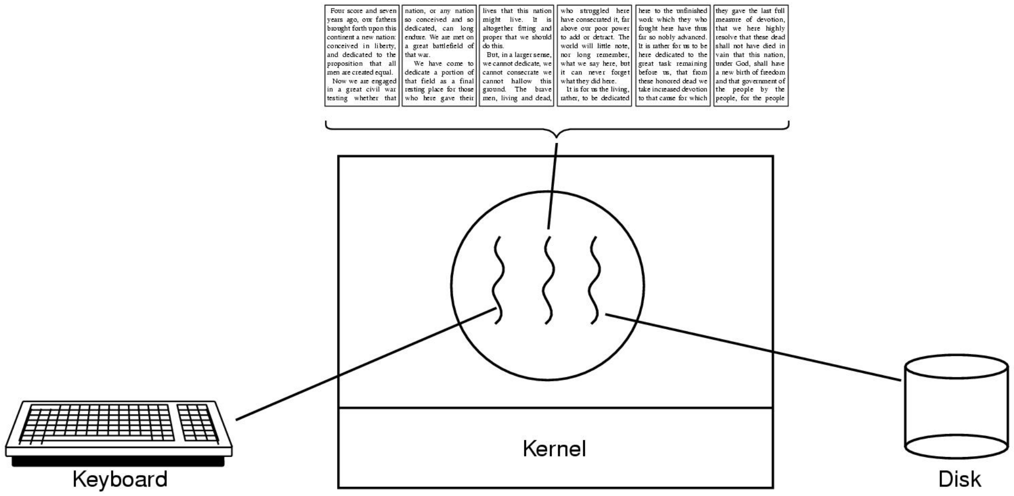


Figure 3: Taken from Tanenbaum

1.2 Thread Usage

When Do Threads Prove Themselves Useful?

- **useful** in applications
  - where **different actions can happen concurrently** (in the same time) $\rightarrow$ **logical parallelism**
  - and use **different "independent" system resources** (e.g. CPU, HDD, etc.) $\rightarrow$ **physical parallelism**
- **NOT useful** in applications
  - using only the single CPU (our considered context)
    - though useful, if multiple CPUs are available
  - in which possible concurrent executions need all the time (compete for) the same single shared resource 5.2.14

Examples of Multi-threaded Apps: A Word Processor 5.2.15

Examples of Multi-threaded Apps: A Concurrent Server 5.2.16

Threads vs. Processes

- which are better and for which kind of applications
- **processes** $\rightarrow$ **isolation**, i.e. totally separated by one another $\rightarrow$ protection
  - $\Rightarrow$ use them when isolation and protection needed
  - e.g. tabs in an Internet browser
  - e.g. document tabs in a PDF reader
  - e.g. Web server handling client sessions
- **threads** (light-wight processes) $\rightarrow$ **resource sharing** (of the same process)
  - $\Rightarrow$ use them when working on the same process' data is needed
  - e.g. parallel, cooperative computations (see examples above) 5.2.17

Questions (1)

On a computing system with **3 logical processors**, specify the **number of threads** needed to **get the best performance** for:

1. checking if N given numbers are prime or not, using an algorithm that divide the N numbers in equal subsets among the threads.
2. calculating the N-th element of the array generated by the rule: x(n) = x(n-1) + x(n-2), x(0)=0, x(1)=1. 5.2.18

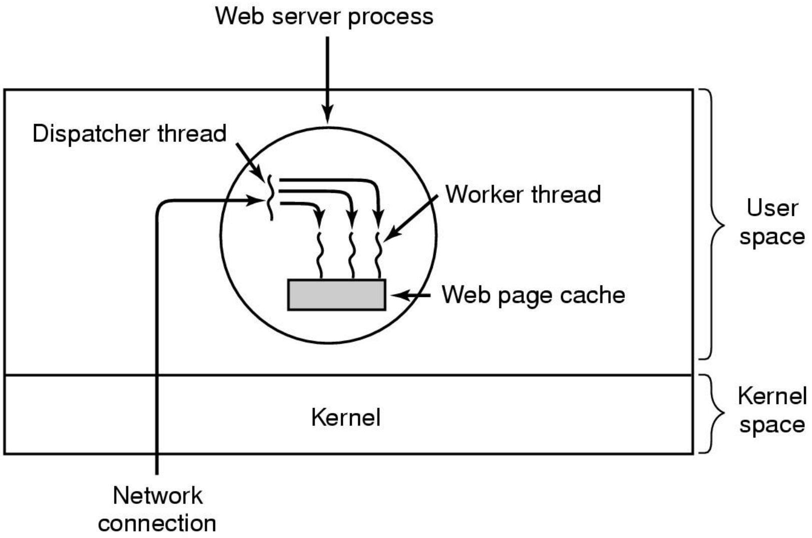


Figure 4: Taken from Tanenbaum

Questions (2)

On a computing system with **1 logical processor**, **a HDD** and **a network card** that can act independently of the system's processors, specify the **number of threads** needed to **get the best performance** for:

1. copying a file from the local HDD to another file on the same HDD;
2. encrypting a file from the local HDD into another file on the same HDD;
3. uploading a file from the local HDD on a remote Web site.

5.2.19

2 Scheduling

2.1 Concepts

Definitions

- **scheduler**: the OS component that **decides**
  - **what process / thread will be run next** and
  - **for how long**
- scheduling policy / algorithm
  - the policy (rules, requirements) / algorithm used by the scheduler
- mainly related to CPU(s)
  - but also consider other system resources
- scheduler's role (motivation)
  - mediates between the more competitors of a limited set of resources
- general classes of processes
  1. compute-bounded (CPU-bounded), i.e. CPU intensive
  2. I/O-bounded, i.e. I/O intensive

5.2.20

Classes of Processes

5.2.21

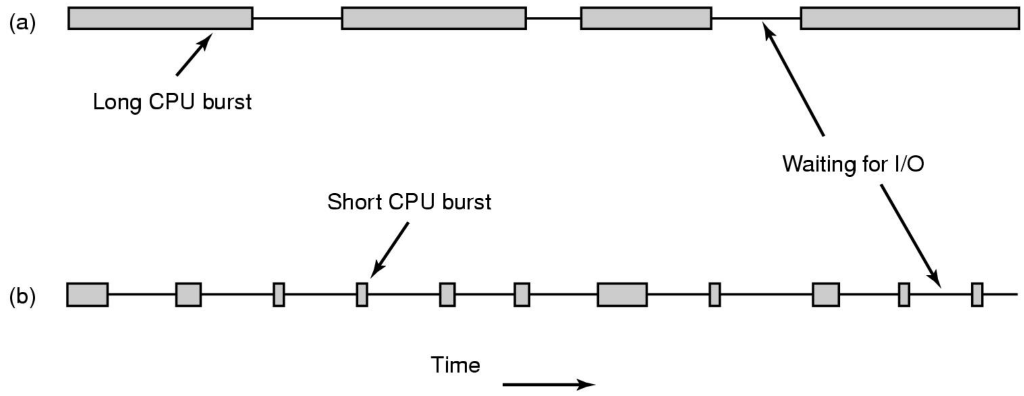


Figure 5: Taken from Tanenbaum

DEMO: CPU-bound vs. I/O-bound Processes/Threads

- create few I/O bound threads, just to see they let the CPU idle a lot (see the percentage they use from CPU)
  - e.g. threads that display very often something on the screen
  - e.g. threads that mostly all the time read something from a file
- create just one CPU-bound thread, just to see that the CPU will be in use all the time
  - e.g. a simple, unrealistic "`while(1);`" thread

5.2.22

Questions (3)

Which threads in the following code are CPU-bound and IO-bound respectively?

```
thread_1 (int n)
{
    int i, x;
    for (x =1, i = 1; i <= n; i++)
        x = x * i;
}

thread_2 ()
{
    char c;
    while (1) {
        read  (0, &c, 1);
        write (1, &c, 1);
    }
}
```

5.2.23

Scheduling Situations

- any time something is changed regarding the CPU's competitors
  - **process creation**
  - **process termination**
  - **process blocking**, e.g. sleeping for a while, waiting for an event etc.
- external events
  - generally, **interrupt occurrences**
  - particularly, the **clock interrupt occurrence**
    - used to implement CPU (time) sharing between processes

5.2.24

Preemptive vs. Non-preemptive Scheduler

- depends on the way the clock interrupt occurrence is used by scheduler
- clock interrupt helps OS keeping track of time passage
- from this perspective a scheduler could be
  - **non-preemptive**
    * lets a process continue its execution until completion/blocking once CPU is given to it
    * i.e. a process could keep CPU if needed
  - **preemptive**
    * suspends a process after a while to switch the CPU to another process
    * i.e. takes the CPU "by force" (yet transparently) from the current process even if it still needs it
    * resumes later the previously suspended process

5.2.25

Categories of Scheduling Algorithms (Examples)

- **batch systems** (processes)
  - do not interact with the users, usually performing long running jobs
  - their users normally expect **reasonable termination time**
  - **fit better non-preemptive scheduling algorithms**
  - or preemptive algorithms with long time quanta
  - reduce the no of process switches $\Rightarrow$ increase overall performance
- **interactive systems** (processes)
  - interact with users
  - their users expect **good response time**
  - **fit only preemptive algorithms**
- **real-time systems** (processes)
  - must perform certain actions in limited time
  - i.e. they must meet their deadlines
  - e.g. react to an external event with a maximum delay
  - **need specific scheduling algorithms** and system characteristics

5.2.26

2.2 Scheduling Algorithms

First-Come First-Served (FCFS)

- it is a non-preemptive algorithm
- simple to understand and implement
- the average waiting time (a.w.t.) depends on the thread order and execution time
  - see Gantt diagram, i.e. graphical illustration of chronological time intervals each thread was scheduled

5.2.27

First-Come First-Served (FCFS). Example 1

- `T1:23, T2:2, T3:3` (`thread : execution time`)
- waiting time for T1 is 0 ms, for T2 is 23 ms, and for T3 is 25 ms
- $a.w.t. = \frac{0+23+25}{3} = 16ms$

5.2.28

First-Come First-Served (FCFS). Example 2

- `T2:2, T3:3, T1:23` (`thread : execution time`)
- waiting time for T1 is 5 ms, for T2 is 0 ms, and for T3 is 2 ms
- $a.w.t. = \frac{5+0+2}{3} = 2.33ms$

5.2.29

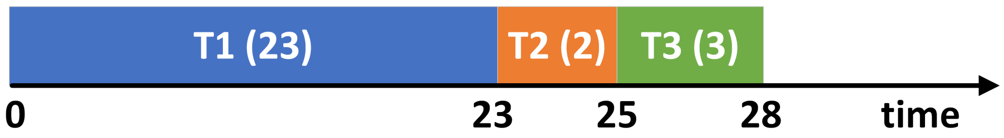


Figure 6: Gantt diagram of process scheduling

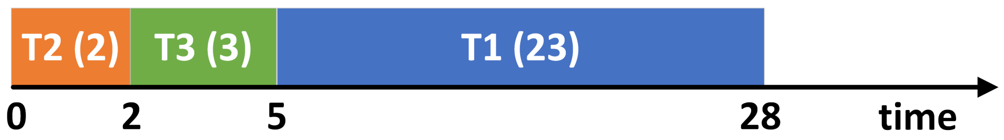


Figure 7: Gantt diagram of process scheduling

Shortest Job First (SJF)

- runtime should be known in advance
- is the optimal algorithm when all the ready threads are available simultaneously
- non-preemptive
  - `0:P1:8,1:P2:4, 2:P3:9, 3:P4:5` (`submit time` : `process` : `run time`)
  - `a.w.t. = (0 + (8-1) + (17-3) + (12-2))/4 = 7.75`
- preemptive (*shortest remaining time next*)
  - `0:P1:8,1:P2:4, 2:P3:9, 3:P4:5`
  - `a.w.t. = ((10-1)+(1-1)+(17-2)+(5-3))/4 = 6,5`

5.2.30

Round-Robin (RR)

- each thread is assigned a time interval
  - time quantum or slice
- it is a preemptive algorithm
- maintain a FIFO list for ready threads
  - like FCFS, but preemptive, based on time quanta
- the length of the time quantum
  - too short ⇒ many CPU switches ⇒ lower the CPU efficiency, i.e. a lot of CPU is wasted for running the context switching code
  - too large ⇒ longer wait times in ready queue ⇒ poor response time for interactive threads
  - 20–100 msec is a reasonable compromise

5.2.31

Priority Based

- each thread has a priority assigned
  - usually a number that describes the thread "importance" (from some perspective)
  - e.g. running time could be such a number: the smaller the higher thread's priority⇒ SJF
- rule: **the ready thread with the greatest priority is always run**
- give a chance also to smaller priority threads
  - i.e. **avoid starvation**
  - by changing periodically the priority of processes
- priority-based policies
  - fixed priorities

- dynamically adjusted priorities
- priority classes
  - there could be more processes with the same priority
  - use priority scheduling between classes
  - use another scheduling algorithm with each class, e.g. round-robin

5.2.32

Questions (4)

Let us suppose that an OS' scheduler uses the **Round-Robin** algorithm with **200 ms** time-quantum. Illustrate on a **Gannt diagram** for the time interval **0 – 1 seconds** the executions of the threads whose code is given below, for the following scenarios.

1.
```
// Thread1              // Thread2

while(1){}              while(1){}
```

2.
```
// Thread1              // Thread2

usleep(300000); // 300 ms
while(1) {}             while(1) {}
```

5.2.33

3 Conclusions

What we talked about

- threads
  - describe multiple, independent executions in the same process
  - share all resources of the process they belong to
- multi-threading applications
- scheduling
  - needed when there are more competitors for a limited set of resources (e.g. CPUs)
  - first-come first-served, shortest job first, round-robin, priority-based

5.2.34

Lessons Learned

Multiple threads in a process are **useful** and **effective** only in particular cases:

1. **logical application parallelism** exists
2. needed **physical resources available**

5.2.35