Programming 3

teodoraristic

September 2021

1 QandA

1.1 Concurrent program?

It contains two or more processes that work together to perform a task. Each one is a sequential program(sequence of statements that are executed one after another). The processes work together by communicating with each other. The communication is done by using shared variables (one writes into a variable that is read by the other process) or message passing (one sends a message that is received by the other). In both cases they need to use sync, there are two kinds - **mutual exclusion and conditional sync**.

- **Mutual exclusion** - ensuring that only one process accesses the critical section at a given moment.
- **Conditional sync** - a process waits for a certain condition to become true.

1.2 Introduce the division of computers according to the simultaneity of commands and data!

Single-core, multi-core, and many-core. So given the number of simultaneous flows of commands (seq of commands that specifies the operations that the computer executes one after another) and data (data flow - seq data that determines over what operations are executed), we divide computers into the following:

1. **SISD - Single Instruction Single Data Stream** - Basic von Neumannov type of computer, sequential computer it executed instructions one by one
2. **SIMD - Single Instruction Multiple Data Stream** - The computer executes one operation over more than one data (This is basically GPU)
3. **MISD - Multiple Instruction Single Data Stream** - Multiple processors and each of them executing its own code but on the same data, this model is not implemented
4. **MIMD - Multiple Instruction Multiple Data Stream** - Multiple processors and each of them executing its own code on same data, can be with shared or distributed memory ; a group of computers connected by a single ...

1.3 What is a process? What's a thread? Describe the differences!

- A **process** is an instance of a computer program that is being sequentially executed by a computer system. Process is started and copied to ready queue, then the scheduler moves the task to execution when a processor is available. Here the process can be stopped because we have I/O operation so it is, when data is ready it is put into ready again, from execution it is put back to ready if the allocated processor time passes, when the code of the process is finished, the process is ended.
- A **thread** of execution is the smallest sequence of programmed instructions and it is usually a component of the process. Thread is basically a lightweight process. It has most of the properties of the process, but it doesn't have its own data. It has its own program counter, stack pointer, and its own registers...
- **The main difference** is that processes are typically independent, where threads exist as a subset of a process. Processes have separate address spaces and threads share their address space. Processes carry a lot more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources.

1 thread works spellcheck,
the other updates the screen.

Application classes (types of applications by how they run processes):
→ multithreaded → # threads > # cores => switch threads on 1 processor (pseudoparallelism)
→ distributed => threads to each comp
→ parallel => # threads = # processors

Types of concurrent computations:
→ shared memory (share a var)
→ distributed memory (msg passing)

1.4 Present the forms of parallel programming!

Note that you have forms, types and concepts (paradigms) of parallel programming.

Forms:

- **Data parallelism** - The parallelization of the process with regard to the number of data to be processed at the same time. Synchronization with barriers is required.
- **Procedural parallelism** - parallelization of the procedure with respect to the number of tasks that can be performed at the same time. CS synchronization is required.
- **The combination of both** → synchronous (uses data & task but it's rare => uses CS sync and barriers)

Types: Hardware architectures: single CPU

- **Programming using shared memory**: Processes share common memory, the communication is done by reading and writing data into a shared variable. Here we need explicit sync because processes can write/read from common variables
- **Distributed memory programming**: processes do not have shared memory, they share a communication channel. The communication is done by sending and receiving messages via the comm channel. Sync is implicit: in order for the message to be accepted, it must be sent

Concepts (book): iterative parallelism, recursive parallelism, produces/consume principle, client/server principle, principle of interaction among equivalent entities (Interacting peers):

- **Iterative parallelism** is used when a program has several, often identical processes, each of which contains one or more loops. Hence each process is an iterative program . The processes in the program work together to solve a single problem; they communicate and synchronize using shared variables or message passing
- **Recursive parallelism** can be used when a program has one or more recursive procedures and the procedure calls are independent, meaning that each works on different parts of the shared data. Recursion parallelism is used to solve many combinatorial problems, such as sorting, scheduling, game playing.

(slika .....)

1.5 Present the producer/consumer principle!

- **Producers and Consumers: Unix Pipes** The producer produces data, so it is a process that computes and outputs a stream of results. The consumer uses the data, so it inputs and analyzes a stream of values. Many programs are producers and/or consumers of one form to another. THE FLOW OF DATA IS DIRECTED FROM THE PRODUCER TO THE CONSUMER, so the consumer process is dependent on the producer process till the necessary data has been produced. Pipelining: each program reads from the input and writes to the output, the program uses the data produced by the previous program and produces data for its successor. A pipe is a buffer(FIFO queue) between a producer and consumer process. New values are appended when a producer process writes to the pipe. Values are removed when a consumer process reads from the pipe. So a producer waits(if necessary) until there is room in the buffer, then writes to the end of the buffer. A consumer waits(if necessary) until the buffer contains a line, then removes the first one. We have implementation using shared variables and various sync primitives (flags, semaphores, and monitors), we can also use message passing.

2. primer: Producer produces tokens, consumers take them and play games; if the buffer is full, don't produce → condition synchronization
if there's nothing in the buffer, can't consume
→ mutual exclusion
oko cons. insert to buffer, turn of producers

1.6 Present the Client/Server principle!

- **Clients and Servers** A client process requests a service, then waits for the request to be handled. A server process repeatedly waits for a request, handles it, then sends a reply . We have a mutual communication: demand, response; A server has many clients, and each client must be handled as an independent unit, but multiple requests can be handled concurrently. So a server can be implemented by a single process that handles one client request at a time, or it can be multithreaded to service client requests concurrently. Present read/write in a file on a shared memory system. Implementation on shared memory machines uses subroutines, programmed using mutual exclusion and conditional sync to protect critical sections and to ensure that the subroutines are executed in a correct order. On the other hand, in a distributed memory system, the server is implemented by one or more processes that usually execute on a different machine than clients. IN BOTH CASES, A SERVER IS A MULTITHREADED PROGRAM, WITH ONE THREAD PER CLIENT.

1.7 Present the principle of interaction between the equivalents!

- **Interaction of equals(Peers)** It occurs in distributed programs when there are several processes that execute basically the same code and that exchange messages to accomplish a task. So we solve a problem by distributing the work among several workers, all of whom perform approximately the same amount of work. In order to solve the problem the workers communicate and cooperate, they use message passing. Interacting peers are used to implement distributed parallel programs, especially those with iterative parallelism.

1.8 What is a critical reference?

- **Critical reference** in an expression is a reference to a variable in an expression that is changed by another process. **At-Most-One-Property**: An assignment statement x = e(expression) satisfies the at most 1 prop if either:

  1. e contains at most one critical reference and x is not read by another process
  2. e contains no critical references, in which case x may be read by other processes.

ako važi at most 1, onda ne treba atomic command

If an assignment statement meets the requirements of the at most 1 prop, then execution of the assignment statement will appear to be atomic.

Critical section is section of a code in which a critical reference / shared variable is changed.

- The sequence of commands that access common variables is called the **critical section** of the program. Critical section of a program can only be executed by one process at a time, so the sequence of critical commands must be performed indivisibly as an atomic command.

kod!!!

- Program's critical section: **entry protocol - CS - exit protocol**. To implement entry and exit protocols we need to ensure: CS & await command are strictly linked

await comand ima 3 tipa: ↓ expensive

- atomic code <S>
- CS entry <S>
- CS exit

atomic operation

- await <B> ↓ bool condition sync most expens.
- await <B> <S> ↓ čekaj da buffer bude prazan pa ga napuni

mutual excl. ↑

nema straig. implement.

  1. **Mutual exclusion of CS** - max one process can enter the CS, perform and exit
  2. **Preventing deadlock** - at least one of the processes competing to enter the CS will be enabled
  3. **Preventing retention of CS** - if one process is waiting to enter the CS, but there is no other process, then this process can begin the CS immediately
  4. **Enabling access to CS** - each process that is waiting to enter the CS will enter eventually

await b → waiting for the process to continue

1.9 Present an atomic command!

- The process executes a sequence of statements. A statement is implemented by one or more atomic commands
- **An atomic command** is a sequence of instructions in the execution of a process, which is performed as an indivisible whole. Examples of atomic actions are uninterruptible machine instructions that load and store memory words.

npr. => gauriši inc counter (u hardveru → load to reg, increment, load to variable)

- Here we can mention that the state of a concurrent program consists of the values of the program variables in a given moment. A concurrent program begins execution in some initial state. Each process in the program executes independently, and as it executes it changes the program state. **CONCURRENT EXECUTION OF PROCESSES IS NON-DETERMINISTIC** (example: If we have a parallel program with n processes and each process executes m atomic operations, then we have (n*m)! / m!. possible paths)

(registri → zamrzni)

svaki thread može da promeni varijablu

$\frac{(n \cdot m)!}{m!^n}$

1.10 Describe the lock principle - locks

- A lock is a logical variable that is true when one of the processes is in the critical section, and it is false when there is no process in the critical section.
- **Lock principle**: The critical section is preceded by an entry protocol and followed by an exit protocol. Basically when a process enters the CS it locks the lock (Jernej's words) and then when it exits it releases the lock. (CS entry protocol locks, CS exit protocol unlocks). Of Course this must be done atomically.

true - locked; false - unlocked

- **Spin-lock**: processes keep spinning in a loop and wait for the lock to open (while true)
- locks with an order: Peterson, Ticket & waiting list (kao Ticket, samo što se daje br. veći od ostalih u sistemu)
- Introduce the atomic commands **TS(Test and Set)** and **FA(Fetch and Add)**. TS: gets a bool variable and stores the value in a temp variable, then it sets the value of the lock to true and it returns the starting value

TS & FA za lock implementation

machine command that's used to set the variable and return the old value to atomically test and set the lock

- TS: if the lock was true, SKIP
- if the lock was false, atomically set to TRUE

we use it bc of race conditions

PRVI dovojci

improvement za TEST & SET is TEST & TEST & SET → bc of memory contention (cache has to be updated)

1.11 Describe the FA command!

- FA: is basically atomic incrementation, the idea is to store the value of the variable, increment the variable and return the starting value of the variable
- The FA command is a machine instruction with the effect of returning the old value of a variable and incrementing it as a single indivisible operation. This comes as a great solution for the ticket algorithm.

1.12 Describe the weaknesses of the TS lock in the loop

- TS: gets a bool variable and stores the value in a temp variable, then it sets the value of the lock to true and it returns the starting value
- A variable lock is shared among all processes, and the while loop writes to the lock variable in each loop. So if each process is running on a separate processor and the lock is stored in the cache of this process (for faster access), after each TS execution, the common memory must be refreshed = **memory contention**

1.13 Describe the Petersen algorithm!

→ lock slobodan + ti si sledeći

- Petersen algorithm, also called the tie-breaker algorithm, is a variation on the critical section that "breaks the tie" when two processes try to enter. It does so by using an additional variable to indicate which process was last to enter the critical section. This is good, because it preserves fairness, because we could have two processes competing to enter the CS, there is no control over which will succeed. In particular one process could succeed, execute the CS, race back around to the entry protocol, and succeed again. So to make it fair, the processes should take turns. This algorithm is simple for 2 processes, but if we have more that 2, things get complicated (idk why)
- If we have n>2 processes we could use the **Ticket algorithm**. This one uses integer counters to order the processes. Basically it gives the process a number (ticket) and the process waits for its turn to enter the CS. Think of it when you go to a bank and take a number and you wait to see your ticket number on the display. So this algorithm uses the atomic command **FA**.

1.14 Describe the conditional variables!

↗ spin lock

- Till now we used the busy waiting to execute the CS, but its weakness is that processes wait in a loop, and this steals processor time. Introducing the blocking wait: Here processes waiting to enter the CS are stopped, they are awakened by another process (a signal is sent). Two mechanisms that use this: conditional variables(cond variable + MUTEX = monitor) and semaphores.
- A conditional variable is a data structure that is made of a queue of stopped processes and operations on this queue that manipulate processes. Basically it is a queue of stopped threads

↗ blocked

- A condition variable is used to delay a process that cannot safely continue executing until the monitor's state satisfies some Boolean condition. It is also used to awaken a delayed process when the condition becomes true.
- Basic operations:
  1. lock a process(the process stops and is put in waiting queue),
  2. send a signal to awaken(resume execution),
  3. unlock (take process from waiting and change status to execute)
  4. check if the queue is empty,
  5. find element with the lowest priority

1.15 Describe operations on conditional variables!

- wait(cv, lock) - release the lock and put the process at the end of a waiting queue.
- signal(cv) - awaken the process that is at the beginning of a queue.
- signalall(cv) - awaken all the processes that are in the queue.

↓ conditional Variable

- empty(cv) - true if the queue is empty

There are few signalization types in conditional variables: signal and continue, signal and wait, signal and urgent wait.

continue immediately

has queue, uses blocking wait

1.16 What is a semaphore?

can store values

- **Semaphores** represent the mechanism for conditional synchronization (red, green light) between processes (trains) in order to ensure mutual exclusion of process access to critical areas (prevent accidents in the critical section of the line).

inc and dec operation

V P

- Semaphore is a data structure that includes an integer variable s, and two atomic operations: **P = try to enter CS and V = release CS**.

2 operations: wait + signal → inc value & return

ATOMIC

dec the value of semaphore if it's >0, if it's 0 = wait

- How to implement the P and V operation in a semaphore? The await command can be implemented by waiting in sleep (eg conditional variables) or by busy waiting (locks). We use the implementation of waiting by stopping the process in semaphores. That's why we use the FIFO queue.

1.17 Present and describe basic types of semaphores.

0 - there is someone
1 - there is noone

1. **Binary semaphore (mutex)** - the value of semaphore can be 0 or 1 and it is used to ensure mutual exclusion of access to critical section and for conditional synchronization. Binary semaphores can also be used to communicate between processes: to signal the state of a particular event or state of a certain condition:
   (a) usually the value of the semaphore 0 signals that the event has not yet occurred, or that the condition is not met
   (b) the process signals (informs) that the event has been executed with operation V - operation V "sends" the signal (to all other processes)
   (c) the process waits for a signal (message) that the event happened, in operation P - the process "receives" the signals in operation P. -**Usage: barriers** - a barrier between 2 processes can be implemented with two binary semaphores, that communicate the arrival and control the exit of the barrier
2. **Composed binary semaphore** - has several binary semaphores. At a given moment only one of the semaphores can be 1, all others are 0. It is used for producer/consumer, and with the use of limited buffer.

↳ the semaphor is true if only one is true

3. **(General) Counting semaphore** - the value of semaphore can be any nonnegative number. It is used for counting how many times a CS is accessed, how many resources are used etc.

blocked until the semaphor is incremented

Mutex je semafor
ne!

Hadoop - allows to send with complicates to others who can commit complications

framework
↑
MapReduce - 3 phases :
(key, value)
↓ ↓
ime br. telefona
name
processes the text counts the # of times it appears

Jak to MapReduce? → It's simple :)
↓
TutorialsPoint yt

- empty(cv) - true if the queue is empty

There are few signalization types in conditional variables: signal and continue, signal and wait, signal and urgent wait.

1.16 What is a semaphore?

- **Semaphores** represent the mechanism for conditional synchronization (red, green light) between processes (trains) in order to ensure mutual exclusion of process access to critical areas (prevent accidents in the critical section of the line).
- Semaphore is a data structure that includes an integer variable s, and two atomic operations: **P = try to enter CS and V = release CS**.
- How to implement the P and V operation in a semaphore? The await command can be implemented by waiting in sleep (eg conditional variables) or by busy waiting (locks). We use the implementation of waiting by stopping the process in semaphores. That's why we use the FIFO queue.

1.17 Present and describe basic types of semaphores.

1. **Binary semaphore (mutex)** - the value of semaphore can be 0 or 1 and it is used to ensure mutual exclusion of access to critical section and for conditional synchronization. Binary semaphores can also be used to communicate between processes: to signal the state of a particular event or state of a certain condition:
   (a) usually the value of the semaphore 0 signals that the event has not yet curred, or that the condition is not met
   (b) the process signals (informs) that the event has been executed with operation V - operation V "sends" the signal (to all other processes)
   (c) the process waits for a signal (message) that the event happened, in operation P - the process "receives" the signals in operation P. -**Usage: barriers** - a barrier between 2 processes can be implemented with two binary semaphores, that communicate the arrival and control the exit of the barrier
2. **Composed binary semaphore** - has several binary semaphores. At a given moment only one of the semaphores can be 1, all others are 0. It is used for producer/consumer, and with the use of limited buffer.
3. **(General) Counting semaphore** - the value of semaphore can be any nonnegative number. It is used for counting how many times a CS is accessed, how many resources are used etc.

1.18 What is a Mutex?

- Mutex is a synchronization primitive that grants exclusive access to the shared resource to only one thread, ensures the mutual exclusion of the implementation of the CS.
- When I am having a big heated discussion at work, I use a rubber chicken which I keep in my desk for just such occasions. The person holding the chicken is the only person who is allowed to talk. If you don't hold the chicken you cannot speak. You can only indicate that you want the chicken and wait until you get it before you speak. Once you have finished speaking, you can hand the chicken back to the moderator who will hand it to the next person to speak. This ensures that people do not speak over each other, and also have their own space to talk.
- Replace Chicken with Mutex and person with thread and you basically have the concept of a mutex.

1.19 Describe MapReduce! Describe the Map phase! Describe the phase Reduce!

MapReduce is a distributed programming model and an associated implementation for processing and generating big data sets composed out of

Shuffle Phase

- **map phase** - takes data (key, value) pairs and produces key, value pairs // The Map function takes a series of key/value pairs, processes each, and generates zero or more output key/value pairs. The input and output types of the map can be (and often are) different from each other.
- **reduce phase** - after all map outputs collect, reduce combines the ones with the same keys // The framework calls the application's Reduce function once for each unique key in the sorted order. The Reduce can iterate through the values that are associated with that key and produce zero or more outputs.

MapReduce: The data is stored on one or more clusters. The files are stored in blocks, the blocks are replicated over the network: the blocks are evenly distributed throughout the cluster

- **reduce phase** - after all map outputs collect, reduce combines the ones with the same keys // The framework calls the application's Reduce function once for each unique key in the sorted order. The Reduce can iterate through the values that are associated with that key and produce zero or more outputs.

1.20 Why MapReduce?

It's simple.

Hadoop can store and process vast amount of information. It has 3 components:
- Storage unit HDFS (splits data into blocks and stores them into blocks on nodes in clusters)
- MapReduce (splits data into parts and processes them independently on separate nodes)
- YARN (resource manager, node manager, application master, container)

1.21 What is Hadoop?

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. Hadoop is an open source implementation of GFS(Google File System) with some limitations.

1.22 What is MPI?

The Message Passing Interface (MPI) is a library of message-passing routines. When MPI is used, the processes in distributed program are written in a sequential language such as C or Fortran; they communicate and synchronize by calling functions in the MPI library.

- MPI: Message Passing Interface: The MPI library is designed to work with distributed processes in multiprocessor or multi-computing architectures, communication between processes with message exchange – API
- Purpose: enable the development of programs for parallel and distributed computing, enable the implementation of distributed programs on various distributed computer
- MPI is the standard for distributed computing.
- Processes can communicate only with processes from the same group, which are handled by one communicator
- Processes communicate with each other using the send/receive functions
- Examples:

1. $\mathbf{MPI}_{B}cast\mathbf{MPI}_{G}ather\mathbf{MPI}_{S}catter\mathbf{MPI}_{R}educe$

Bcast, gather, scatter, reduce

1.23 Monitors

In concurrent programming, a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false

Monitor is the data structure (class) that is used:

2. to perform atomic operations and commands over common variables that are encapsulated in the monitor.
- to control access to shared resources outside monitors.
- Later we will look at the use of monitors for various synchronization techniques

1.24 Barriers

Barrier is a point in the program that all processes must reach in order for the program to continue