MIT World Peace University

Computer Networks
Second Year B.Tech Semister 3
Academic Year 2022-23

Operating Systems

Notes from Tananbaum and Classes

Prepared By

P34. Krishnaraj Thadesar

Batch A2

October 12, 2022

Contents

1 Concurrency Control - Waht to learn

P rocess Synchronization: Principles of Concurrency, Requirements for Mutual Exclusion: Hardware Support, OS Support

C lassical Synchronization Problems: Readers Writers Problem, Producer and Consumer Problem

P rinciples of Deadlock, Deadlock modelling, prevention, avoiance and stuff.

When 2 processes are running at the same time, when they are interdependent on each other through inputs and outputs, then you would call that concurrency. We arent doing multiprocessing by choice, here we just gotta do it at the same time somehow.

2 Design Issues in concurrency

1. Communication among processes
2. Sharing and Competition for resources
3. Synchronization of activities of multiple processes
4. Allocation of processor time to processes.

3 Contexts of Concurrency

1. Multiple Aplications like Multiprogramming(diff programs on a single processor) and Multiprocessing(on diff processors)
2. Strucctured Applications: Some applictions can be effectively programmed as a set of concurrent proesses. (Principles of modular design and structured programming)
3. OS Structure: OS often implemented as a set of processes or threads.

4 Key terms related to concurrency

1. Atomic Operation: A sequence of one or more statements that appear to be inivisible that is no other process can see an intermediate state or interrupt the operations
2. Critical Section: A section of code within a process that requires access to shared resources and that must not be executed while another process is in corresponding section of code.
3. Deadlock: A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
4. Mutual Exclusion: The requirement that when one process is in critical section that access shared resources, no other processes may be in critical section that accesses any of those shared resources.
5. Race Condition: A situation in which multiple threads/Processes read and write a shared data item and final result depends on relative timing of their execution.
6. Starvation: A situation in which a runnable processes is overlooked indefinitely by the scheduler, although it is able to proceed, it is never chosen.

4.1 Race Condition

A race Condition occurs when multiple competing processes or threads read and write data items so that final result depends on the order of eecution of instructions in multiple processes. so 2 processes p1 and p2, say they share global variable 'a'. P1 updates a to 1, and p updates it to 2. Thus two tasks are in a race to write variable 'a'. Loser of race is the one that determines the value of 'a'.

5 Difficulties due to concurrency

1. Sharing of global resources: Eg. Two processes both make use of global variable and both perfrom read and write on that variable, in which read and write are done is critical.
2. Management of resources optimally. Eg. Process has gained the ownership of IO devices but is suspended before using it, thus locking IO device and preventing its use by other processes.
3. Error locating in Program: Pesults are not deterministic are reproducible.

5.1 Example

```
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

1. Uniprocessor multiprogramming, single user environment
2. Many applications can call this procedure repeatedly to accept user ip and display on screen.
3. User can jump from one applciation to other.
4. Each application needs or uses procedure echo.
5. Echo is made shared procedure and loaded into a portion of memory global to all applications
6. Single copy of echo procedure is used, saving space.
7. When echo procedure is invoked by some process, an dif fthe process gets suspended for any reason beffore completing it, then no other process can invoke echo till prcess that was suspended gets resumed, and completes echo. Thus other processes are not allowed to access it.
8. It would make more sense if you visualize this in terms of a multi user operating system. So if many people are using a server kinda thing at the same time.

5.2 Example

Say you have global variables that hvae values b = 1, c = 3 If P1 executes b = b + c
P2 does c = c + b

Now if P1 executes first, you would get b = 3, c = 5;
If P2 was to Execute first, then b = 4, c = 3

This is a problem.

5.3 Process Interaction

- Processes that are unaware of each Other (Competition)
    1. Here Multiprogramming of Multiple indepent Processes
    2. OS Needs to know about competition for resources such as printer, disk, file etc.
    3. Potential Problems: Mutual Exclusion, Deadlock, starvation
- Processes that are indirectly aware of each Other (Cooperation by sharing)
    1. Shared Access to some object such as shared variable.
    2. Cooperation by sharing.
    3. Potential Probelm: Mutual Exclusion, Deadlock, starvation, data coherence etc.
- Processes that are directly aware of each other (Cooperation by communication)
    1. Cooperation by communicatio, communication primitives are available.
    2. Potential control problems: Deadlock and starvation.
    3. Mutual exclusion not a probelm, as both are aware of each other.
    4. DeadLock is possible, and so is starvation.

5.4 Three Control problems

- The need for mutual Exclusion: Two or more processes require access to single non sharable resources such as printer, say. Such a resource is called a critical resource, and the portion of code using it is called as critical section of the program. Mutual Exclusion is when you allow only process to work when its in its critical section.
- A common problem would be deadlock, where say P1 is using the printer, and P2 is using a file, Now P1 has to wait for P2 for using the file, and P2 has to wait for P1 to use the printer.
- Starvation would be like how we schedule the processes, and so periodic access would be given. While one process is using some resource, others have to patiently wait, and so they are starving.

Remember to Examples a whole lot while answering questions of OS

5.4.1 Cooperation among processes by sharing

- Processes that interact with other processes without begin explicitly aware of them, access to shared variables, or files or databases is what is being talked about here.
- Processes may use and update shared data without reference to other processes but know that other processes may have access to same data.

- Processes must cooperate to ensure that the data that they share are properly managed.
- control problems of mutual exclusion, deadlock and starvation are again present.
- Data Items are accessed in 2 Modes: Reading and Writing.
- Writing operations must be mutually exclusive.

5.4.2 Requirements for Mutual Exclusion

- Any facility or capability providing support for mutual exclusion should meet following requirements:
- Mutual Exclusion must be **enforced**. Only one process at a time in the CS, among all processes that have CS for same resource or shared object.
- A process that **halts in its non CS** must do without interfering with other processes.
- A Process requiring access to CS must not be **denied or delayed** indefinitely. So no deadlock or starvation.
- When no process is in its CS, any process that requests an entry to its CS, ust be permitted to enter without delay.
- No Asusmptions are made about the relative speed of the process.

5.4.3 Approaches to satisfy the requirements of Mutual Exclusion

1. Hardware Approaches
2. Support from the OS or programming language itself.
3. Software Approach (No support from OS or programming language) You have to write it yourself.

5.5 OS or programming language support

1. Semaphore : It is an integer variable that sovles critical section problem, after initialization. Its value can only be accessed by 2 atomic instructions.
   (a) wait(s)

```
wait(s) {
  while(s<= 0) {
    s --;
  }
}

```

   (b) signal()
   (c) *Internal Working*: Consider if P1, P2, P3, P4 etc are processes that want to go to the critical section.

```
        do{
          wait(s);
          // goes in critical section.
          signal(s);
          // remainder section;
        } while(true);
```

Explicitly you cant change the value of semaphore.

2. There are 2 types, Binary and Counting Semaphore.
   - Mutex: Mutual exclusion
   - Progress
   - Bounded Wait
3. Mutex
   -
4. Monitors

5.5.1 Problem

Suppose we want to synchronize two concurrent processes P1 and P2 using binary semaphores s and t. The code for the processes P2 and P2 is shown below.

```
Process P1:
while(1)
{
w: ____
print '0'
print '0'
x: ____
}
```

```
while(1)
{
y = ___
print(1)
print(1)
z = ___
}
```

The required output is "001100110010011" Processes are sunchronized using P and V operations on semaphores s and t. Choose the correct options from the following at points w x y and z. While of the following option is correct?

1. P(s) at w, V(s) at x, P(t) at z, s and t initially 0
2. P(s) at w, V(t) at x, P(t) at y, V(s) at z, s initially 1 and t 0
3. P(s) at w, V(s) at x, P(t) at z, s and t initially 0
4. P(s) at w, V(s) at x, P(t) at z, s 1 and t 0

6 Producer Consumer Problem

- it is one fo the classic problems of Synchronization
- Producer produces an item and adds to a buffer of limited size (bounded buffer)
- Consuder takes out an item from buffer and consumes it.
- Buffer is a shared resource and must be used in a mutual exclusion manner by processes.
- Producers to be prevented from adding into a full buffer.
- Consumers to be stopped from taking out an item from an empty buffer.
- To solve this you have to use the semaphores. And wait and signal will be used.
- A Cooperative Process: The Processes which are sharing the resources like code, memory, data item, and stuff like that.

```
// Here count is used to count number of items present in the buffer.
// Assume that the buffer has like 8 items.
// initially obviously the value of count would be 0;

int count = 0;
int in = 0; // it is the position where the producer is going to write the newly generated data item.
void Producer(void)
{
  int itemp;
  while(true)
  {
    Produce_item(itemp);
    while(count == n)
    {
      buffer[in] = itemp;
      in = in + (1 mod n); // to
      count = count + 1; // this is then again split into like 3 other commands in assembly.
      // The micro instructions for this line are (in assembly)
      // load Rp, m[count]
      // incr Rp
      // load m[count] Rp
    }
  }
}
void consumer()
{
  int itemc = 0;
  int out = 0;
  while(true)
  {
    while(count == 0)
    {
      itemc = buffer[out]
      out = out + (1 mod n)
      count = count - 1;
      Consume_item(itemc)
    }
```

```
    }
}
```

6.1 Solution to producer cnsumer problem using semaphore

- With a Bounded Buffer, The bounded buffer producer problem assumes that there is a fixed buffer size, in this case the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

```
\\ initialization

char item; // could be any data type.
char buffer[n];
semaphore full = 0; // counting semaphore for full slots
semaphore empty = 1; // counting for empty slot
semaphore mutex = 1; // binary sema for mutual exclusion of buffer.
char nextp, nextc;


```

```
  // Producer
  do
  {
    produce and item in nextp
    wait(empty); // wait until empty > 0 and then decrement 'empty'
    wait(mutex); // acquire lock
    add nextp to bufer // add data to buffer
    signal(mutex); // release lock
    signal(full); // increment full
  } while(true)


  // Consumer
  do{
    wait(full);
    wait(mutex);
    remove an item from buffer to nextc // remove data from buffer
    signal(mutex);
    signal(empty);
    consume the item in nextc
  } while(true);

```

6.2 Reader and Writer Problem

- If you are running concurrently, Reader and Writer together would be a problem
- Writer and Reader, this would also be a problem
- Writer and Writer, this would be a problem
- Reader and Reader, this wont be a problem

```
semaphore db = 1;
semaphore mutex = 1

void reader()
{
  semaphore rc = 0; // reader count
  while(true)
  {
    down(mutex); // this is like wait, coz you are decrementing anyway
    rc = rc + 1
    if(rc == 1)
    {
      down(db);
    }
    up(mutex);

    // read the data

    // Exit section
    rc = rc - 1;
    if(rc == 0)
    {
      up(db)
    }
    up(mutex);
  }
}

void writer()
{
  while(true)
  {
    down(mutex);
    // do something or write
    up(db);
  }
}
```

7 Inter Process Communication

There are several mechanisms for inter process communication (IPC)

- Signals
- FIFOS that are called pipes.
- pipes
- Sockets
- Message passing
- Shared memory
- Semaphores.

7.1 Pipe

A Pipe is a communication medium between two or more related ro interrelated processes. It can be either within one process ro a communciation betwen the child and the paretn processes. UNIX deals with pipes the same way it deals with files. A process can send data down a pipe using a write system call and another process that can receive the data by using read at the other end.

- Within programs a pipe is created using a system call named pipie.
- This system call would create a pipe for one way communication.
- This call would return zero on success and -1 on case of failure.
- If successful this cal returns two file descriptors.

```
1    #include<unistd.h>
2    int pipe(int filedes[2]);
3    // filedes is the file handler.
4
```

- filesdes is a two integer array that that will hold the file descriptors that will indentify the pipe if successful.
- filedes[0] will be open for reading from the pipe.
- filedes[1] will be open for writing down it.
- piep can fail returns -1 if it cannot obtain the file descriptors (exceeds user limit or kernel limit)
- console > parent > write > read by child > console > display > parent.

8 Monitor

- Monitors are a synchronizatiion construct.
- Monitors contain data variables and procedures.
- Data variables cannot be directly accessed by a process.
- Monitors allow only a single process to access the variabels at a time.

9 Deadlock

As mentioned before, its whne a finite number of resources are to be distributed among a number of competing processes, then this happens. A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

9.1 Deadlock Characterization

Deadlock would happen if some of these things happen at the same time.

- Mutual Exclusion: Resources must be held in a nonsharable mode.
- Hold and Wait: A process holding at least one resource is waiting to acquire additional resources held by other processes.
- No Preemption: A resource can be released only voluntarily by the process holding it. after that the process has completed its task.
- Circular wait: There exists a set of waiting processes such that P0 is waiting for a resoruce that is held by P1. Who is then again waiting for a resource that is hed by P2, and so on, until the last process being dependent on P0 completing its task, which means no one is going to do anything.

9.2 Resource Allocation graph

- If there isnt a cycle, then there there is no deadlock.
- If graph has a cycle, then if only one instance per resource type, then deadlockk.
- If several instances of the resource, then chance of deadlock.

9.3 Method of Handling a Deadlock.

Ensure that the system will never enter into deadlock.

- You can either prevent it, naturally
- or you can avoid it, which i dont even get like arent they the same thing?
- By ensuring that atleast one of the conditions cannot hold, we can prevent deadlock.
- Like say **Mutual Exclusion**, Things like printers arent sharable by nature. so we better not allow a 2 processes to use the printer at the same time.
- Holding and waiting is another way to avoid deadlock. Require the process to request what resoruce it needs, and allocate it all the resources before it begins execution. If there is, Low Resource Unitlization, so starvation is also possible.
- Another problem would be **no preemption**: so if a process that is holding some resoruce and requests another resrouce, that cannot be immediately allocated to it, then all resoruce currently being held are released.
- Preempted resourcees are added to the list of resrouces for which the process is waiting.
- Process will be something
- **Circular Weight**: Impose a total ordering of all resource tyes and require that each process requests resources in an increasing order of enumeration.

Allow the system to enter a deadlock state and then recover. Ignore the problem and pretend that deadlock never occured in the system. Used by most Operating systems.