Concurrency

Operating System Lab-in-a-Box

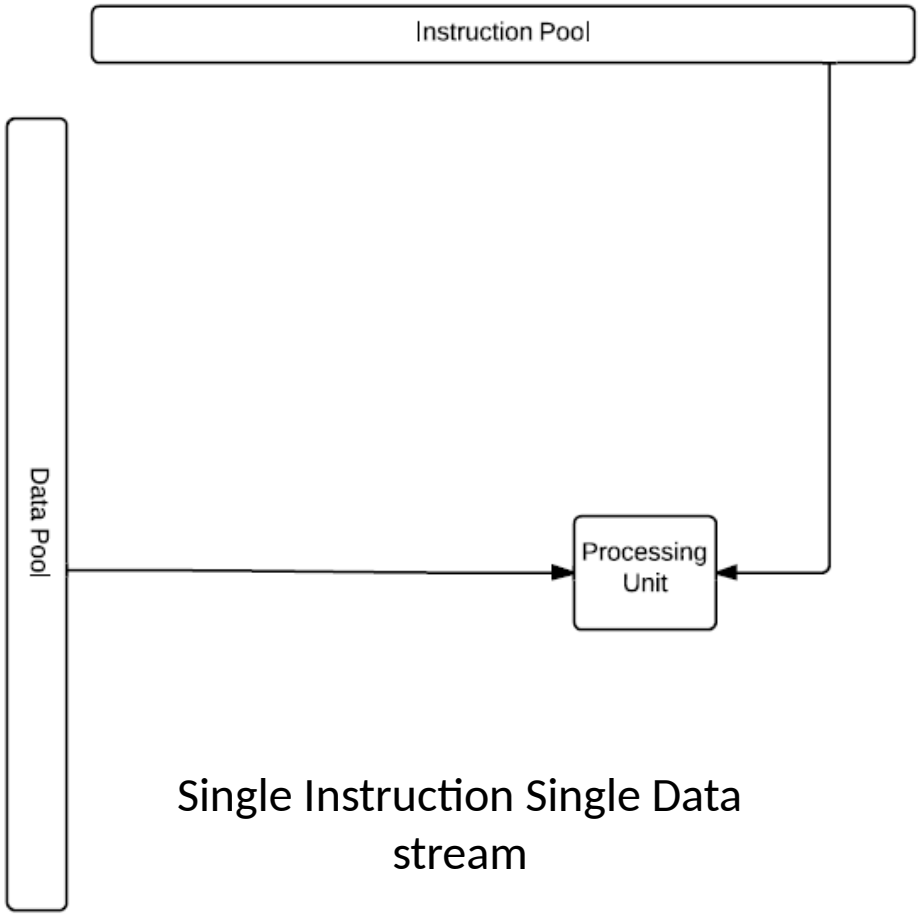
Previous

- Scheduling
- Assignment
  - Quantitative analysis and comparisons of various schedulers

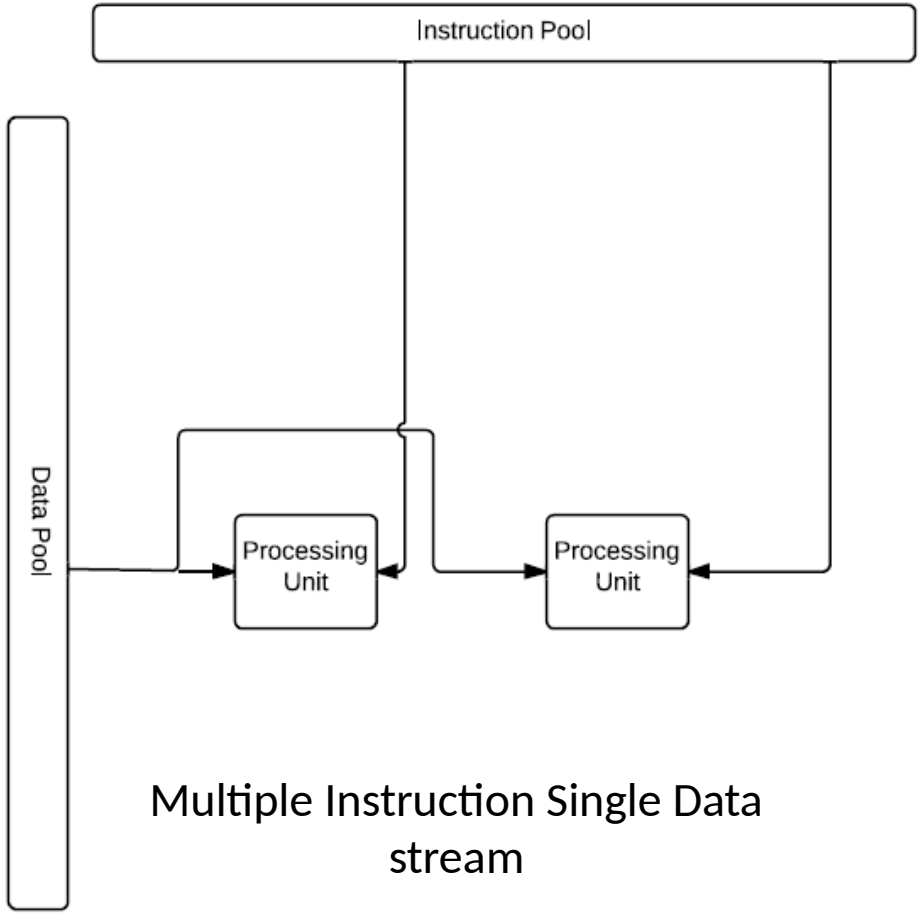
Current Module

- Concurrency Scheduling
- Multiprocessing environment
- Read-write by multiple CPUs and consistency problem
- Solutions with Mutual Exclusion, Hardware Mutex, Software Mutex
- Example: Dekker’s algorithm
- Use of Semaphore and preventing busy waiting
- Message passing and Mail box for communication
- Deadlock and Solution to avoid it.

Multiprocessing and Flynn's Taxonomy

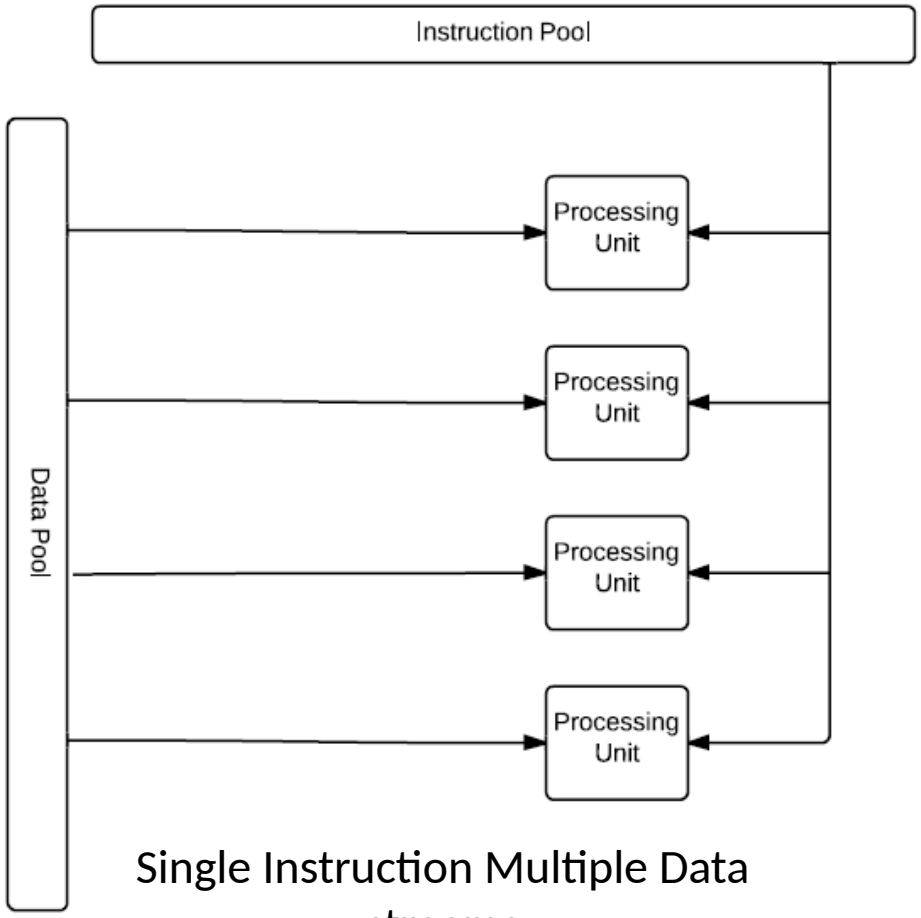


Single Instruction Single Data stream

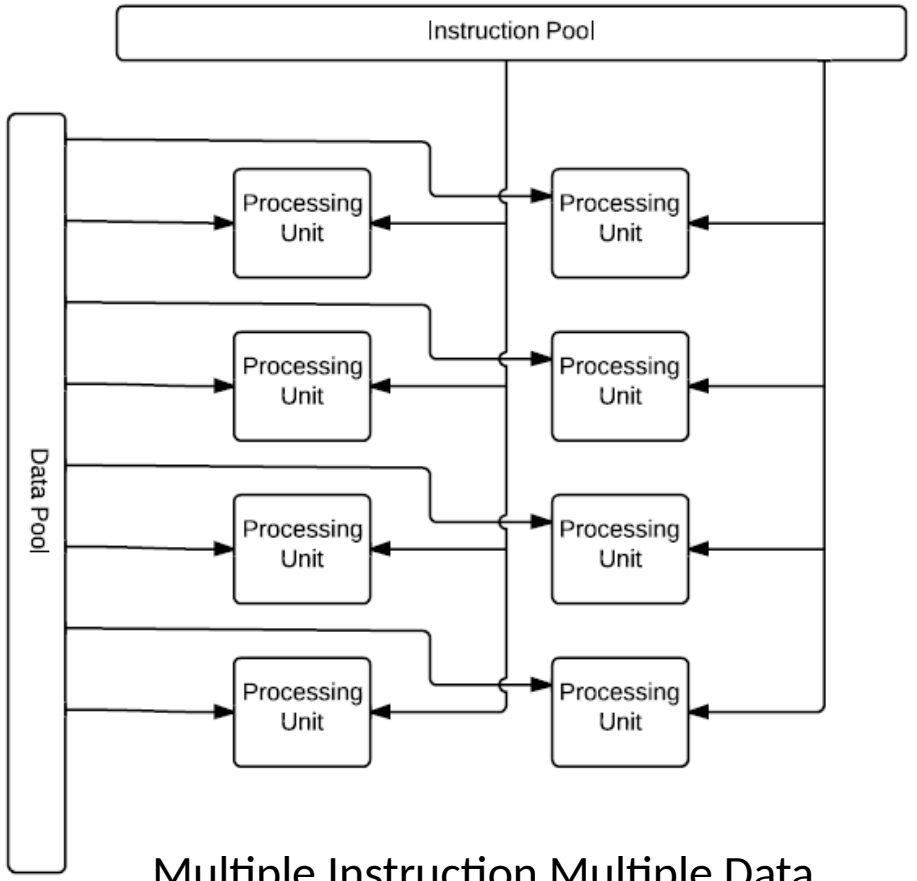


Multiple Instruction Single Data stream

Multiprocessing and Flynn's Taxonomy



Single Instruction Multiple Data streams



Multiple Instruction Multiple Data streams

Multiprocessing and Flynn's Taxonomy

- SISD
  - Sequential without parallelism
  - Concurrency at hardware level
- SIMD
  - A single program on many processing units e.g. Graphics Processing Units (GPUs), vector processors
- MISD
  - Rarely used
- MIMD
  - Many processors executing different instructions on different data
  - Distributed systems
  - Shared memory

Operating Systems on Multiprocessors

- OSs on shared memory multiprocessors are different from those on uniprocessors
  - Centralized or symmetric: symmetric multiprocessing (SMP)
  - Cache organization: Shared or private L1? L2?
  - Memory organization
  - Scheduling – concurrency between processes, which processor, dynamic/static
  - Scheduling – concurrency between threads, one process on multiple processors, load sharing or gang scheduling?
  - Kernel and program: re-entrant?

Concurrency

- Uniprocessor – interleaved processes
- More problems as a result of multiple processes running simultaneously (overlapping)
  - More sophisticated – same abstraction as uniprocessor?
    - Can simple abstraction encapsulate all the features?
    - How do programmers modify their programs to accommodate the proper abstraction?
  - Resource management – share resources in a safe and optimized way
    - Can a resource always be shared? – mutex and deadlock
    - Potential dependency
    - How to share?
    - Room for improvement and optimization?
  - Evaluation and debugging – unpredictable behaviour
- Are all these tasks for the OS only? Can hardware, application software and users help?

The Read-Write Problem Caused by Sharing

- Global variable *shared = 0* at the beginning
- Two processes in parallel:
  - *temp1=shared+1;*
    shared =temp1;
  - *temp2=shared+2;*
    shared=temp2;
- What is the value of *shared* at the end?
  - *1?*
  - *2?*
  - *3?* (Sequential)
  - What do you expect from running this sort of program?

Mutual Exclusion

- Mutually exclusive resources such as *shared* variables, or printers (a hardware device)
- Race condition – non-deterministic, depends on the execution sequence of processes
- Mutex controls access to *shared* resources, you get the expected value as if processed by a uniprocessor system
- Enforced to ensure only one thread of execution can have access at any time
- Critical Section – the code that accesses the mutually exclusive resource
- Requirements for good mutex:
  - Enforced
  - Mutual
  - The only reason a request to a critical section is rejected/delayed is that another process is accessing already
  - Process can only access the critical section for a finite time
  - No deadlock and starvation

Solutions

- Competition for the resource(s)
- The operating system (or other form of intermediary) has to deal with the problem

- A mutex can be implemented on both, the hardware level and the software level
  - Hardware – when something is in the critical section, disable all possible interrupts
  - Software – busy-waiting is the common problem, also deadlock and starvation
- Also OS services:
    - Semaphores
    - Message Passing

Hardware Mutex

- For a uniprocessor system, it is sufficient to just disable interrupts! Common technique used in embedded software
- What about multiprocessors?
  - Atomic instructions – indivisible operations
    - Read/write to memory is always atomic
    - Based on testing the states of memory: test-and-set, compare-and-swap
    - Busy waiting
    - Cannot prevent deadlock and starvation
      - 2 Locks and 2 processes
      - More than 3 processes
      - Hardware alone is clearly not enough
      - Helpful and provides support for software approach

```
while(true){
        while(compare_and_swap(lock,0,1)==1
        /* critical section*/;
        lock=0;
        }
}
```

Software Mutex

- How to detect the state of the resource? - flags
- Examine the flags before entering a critical section, if not met then wait
- Clear the flag before leaving
- The problem becomes what sort of flags to set and how to manage the flag
- Try yourself
  - Without causing deadlock or starvation
- Needs hardware atomic instructions support
  - Out-of-order execution
  - Memory barrier instructions

- Some well-known algorithms:
  - Dekker’s algorithm – 1965 – First known correct solution to the mutex problem
  - Lamport's bakery algorithm – 1974

Dekker's algorithm

```
begin integer c1, c2 turn;
    c1:= 1; c2:= 1; turn = 1;
    parbegin
    process 1: begin A1: c1:= 0;
            L1: if c2 = 0 then
                begin if turn = 1 then goto L1;
                    c1:= 1;
                B1: if turn = 2 then goto B1;
                    goto A1
                end;
            critical section 1;
            turn:= 2; c1:= 1;
            remainder of cycle 1; goto A1
        end;
process 2: begin A2: c2:= 0;
            L2: if c1 = 0 then
                begin if turn = 2 then goto L2;
                    c2:= 1;
                B2: if turn = 1 then goto B2;
                    goto A2
                end;
            critical section 2;
            turn:= 1; c2:= 1;
            remainder of cycle 2; goto A2
        end
    parend
end .
```


Dijkstra, E. W. "Cooperating Sequential Processes, manuscript, 1965." – The original paper that introduces the Dekker's algorithm

Semaphore

- In the same paper, Dijkstra provided another approach that allows concurrency control over multiple identical exclusive resources – semaphore
- An integer variable *s* (or a structure) records the number of currently available resources
- The initial value of *s* – number of copies of the resource

- Two operations can modify the variable
  - *wait(s)* – originally *P* (*proberen*) by Dijkstra: request of resource, decrement *s* and only grant access if *s* is positive, otherwise blocked
  - *signal(s)* – originally *V* (*verhogen*) by Dijkstra: release of resource, increment *s* and dispatch the next blocked process

An Example

- Three public bikes are provided to students
  - You need to ask for a key at the front desk
  - After use, return the key and bike
- Can the front desk use a semaphore to manage the bikes?

```
int key =3;
……
void student(){
        wait(key);
        /*critical section*/
        signal(key);
}
```

- Correct?
  - Are the keys/bikes identical?
  - Yes, then correct
  - What if they’re not? Use mutex instead

Semaphore

- Semaphore does not keep track of the order of access – effectively treating all resources identically
- How do we schedule the blocked processes?
  - Strong semaphore: FCFS queue - Dijkstra
  - Weak semaphore: randomly decide the next waiting process – starvation!
- Binary semaphore: $s \in \{0,1\}$ or boolean type
  - Possible to make a general semaphore by using a private integer variable and two binary semaphores
  - The solution is also given in the original paper by Dijkstra

How Do Semaphores Prevent Busy-Waiting

- The busy-waiting semaphore is sometimes referred to as a spinlock
  - Sometimes useful in SMP
  - Avoids context switching if just waiting for short time
- Or block the process if there is no available resource!
  - *Wait()* – if *s* = 0 – Event Wait
  - *Signal()* – Event Occurs
  - Need a queue or a list

Producer-Consumer Problem

- A common application for semaphores is discussed in the original paper by Dijkstra.
- Single buffer
  - The implementation of the buffer does not matter
- Two cyclic processes (could be more than two):
  - Producer produces a portion of information and adds (writes) that to the buffer
  - Consumer takes (reads) a portion of information from the buffer and consumes it
- Two problems:
  - Exclusive access to the buffer
  - Buffer underflow or overflow (bounded buffer, also discussed in the paper)

Producer-Consumer Problem

```
begin integer number of queuing portions,
buffer manipulation;
    number of queuing portions:= 0;
    buffer manipulation:= 1;
    parbegin
    producer: begin
        again 1: produce next portion;
            P(buffer manipulation) ;
            add portion to buffer;
            V(buffer manipulation);
            V(number of queuing portions);
            goto again 1
        end;
    consumer: begin
        again 2: P(number of queuing
portions);
            P(buffer manipulation);
            take portion from buffer;
            V(buffer manipulation);
            process portion taken;
            goto again 2
        end
    parend
end
```


Dijkstra, E. W. "Cooperating Sequential Processes, manuscript, 1965.“ – The original paper that introduces the Dekker’s algorithm

Message-Passing/Mailbox

- Alternatively, the operating system provides the following primitives:
  send (destination, message) – usually nonblocking
  receive (source, message) – usually blocking
- As an aid to exchange information, they may be used in synchronisation and building mutex
- In addition to the shared memory case, also works on distributed systems
- Mutex:
  - A token message, a process either has it or is waiting for it
  - Who ever owns the token can access a critical section
  - Return the token to the mailbox after use
- Producer-consumer problem:
  - TCP

```
while (1) {
        receive (mailbox, token);
        /* critical section */;
        send (mailbox, token);
}
```

More problems…

- Readers-writers problem
  - Priority?
- Dining philosophers problem
  - Cigarette smokers problem

Deadlock

- Imagine all the philosophers pick their left-hand side fork at the same time. What would happen?
  - Everyone waits for their right-hand side philosopher to release their fork
  - While keeping their own fork
  - Cold war on the dining table – deadlock

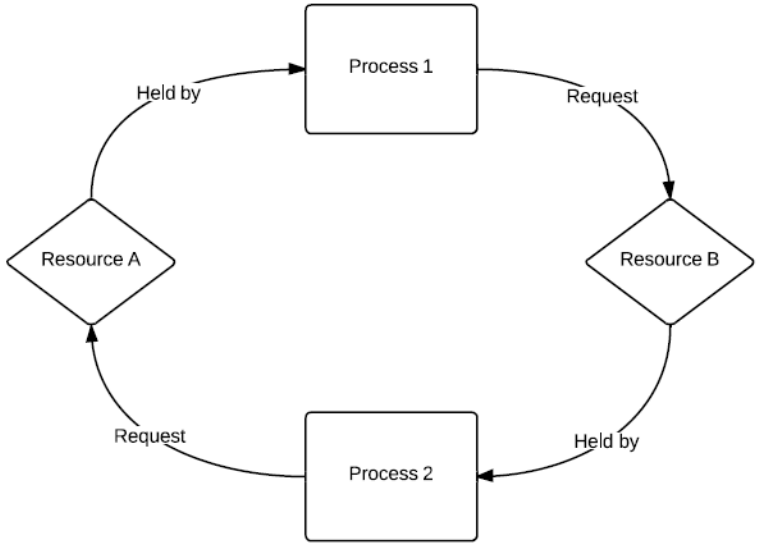
- All processes are waiting for others to finish so that required resources will be released, while reluctant to give up the resources that are also required by others. Thus nobody ever finishes.

- Generally, no perfect solutions yet.

Coffman Conditions for Deadlock

- If the following conditions are all true, deadlock may occur:
  - Mutex: exclusive resource, non-shareable
  - Resource holding: request additional resources while holding one
  - No preemption: resource can not be de-allocated or forcibly removed
  - Circular wait: circular dependency or a closed chain of dependency

necessary but not sufficient conditions



Dealing with Deadlock

- Ostrich algorithm
- Deadlock prevention: if any of the Coffman conditions are false
  - Mutex is inevitable
  - Request all resources at the beginning– either pick two forks at the same time or wait / all-or-none
  - Preemption is inevitable
  - Prevent circular wait condition: Resource hierarchy solution again by Dijkstra
    - The only practically avoidable condition
- Deadlock avoidance
  - Evaluate the chance of deadlock while allocating a resource. Grant or deny based on this information
  - Banker's algorithm
- Deadlock detection: what to do with the existing deadlock?
  - Model checking
  - Kill all or part of the deadlocked processes?
  - Resource preemption?
  - Restart? Watchdog for embedded system

Banker's Algorithm

- Assume the following is known
  - 1.Max possible resources requested
  - 2.Resources allocated
  - 3.Resources still available

	Max			Allocated			Need			Avaliable		
	A	B	C	A	B	C	A	B	C	A	B	C
Process1	3	3	1	1	1	1	2	2	0	1	2	1
Process2	0	1	2	0	0	1	0	1	1			
Process3	1	2	2	0	1	0	1	1	2			

  - Find the process with the fewest currently needed resources, which is also fewer than currently available resources
  - Grant Process 2's request, then Process 3's and finally Process 1's
  - Possible to have no solutions! (safe/unsafe)
  - Practical?

Next

- Lab
- Memory