Next week Tues: Revision session  
Notice board for ST1 detials (only chp1-3, YAY)

Tomorrows prac is functions

**Critical Section problem**

We can handle critical section if we allow processes to get mutual exclusion permissions. If one process is in critical section, you can stop another process from going in that section.

-Entry section: process must ask permission to enter critical section (EVALUATE)  
-Exit section: Allow other processes to now enter critical section  
-Remainder section: Whatever happens after

**A solution to the critical section problem must have:**

1. Mutual Exclusion: Only one process can be in critical section at a time.

2. Progress: Processes can work amongst themselves to select which should go to section (when empty)

3. Bounded waiting: Make sure that a process doesn’t exit, then immediately enter section again. Where a process waiting never enters section as always waiting (don’t want).

**Critical section handling in OS**

**Preemtive:** Can interrupt process

**Non-Preemtive:** While doing something, can continue doing something till voluntarily go somewhere else (ready queue)

**X,Y,Z Solutions – quite a few**

These are good ways to show aspects you must look at when solving issue.  
We wont be looking at these really.

**Petersons solution**: Assumes Loading and storing of variables is atomic

Atomic instructions: Instructions you can execute that you are guaranteed will not be interrupted. (**NBNB terminology alert)**

**Turn:** variable indicates whose turn it is to enter section  
**Flag array:** Shows if a process is ready/wants to enter section

**Synchronization hardware**

**-This issue was found and solved by hardware manufacturers eg Intel**

1. You create a locking mechanism supported in hardware.  
When lock enables, no process can go in that section.  
When you exit, unlock.  
--Compared to public bathroom stalls.

**Below 2 also use locking**

2. Disable interrupts – works with UniPorcessors (one processor)

What happens if you go into an infinite loop, while in critical solution? Interupts are disabled, so cannot ever exit.  
This is why this is normally a bad idea – stops the working of whole OS.

3. Build in atomic hardware instructions

-One line of code, that can exchange the value of 2 variables (normally need 3 lines of code to do such)

**Text book goes into high level of eg test and set principle:** We can just read through, not very important.

Assembly instruction: Exchange

**Mutex Locks**

There are software processes that allow the setting and releasing of locks.   
-acquire() lock, go into critical area  
-release() lock, exit critical area  
--These calls must be atomic

**Busy waiting/Spinlock** – where an instruction is waiting for critical section to be released, constantly seeing/checking if critical section is available.  
-Benefit: Simple to implement.  
-Bad: Wasting processing time, as the process busy waiting cannot make progress during this state  
**The above is what we use to identify mutex locks**  
  
c++ **Code**:

Mutex abcVariable  
int abc  
  
abcVariable.lock()

Code between these 2 cannot be interrupted… only modified by 1 thread at a time  
eg abc++

abcVariable.unlock()

**Semaphore**

**Busy wait semaphore**: Instead of using a flag with true or false, use a counter. While integer negative or zero, spin.

-wait()  
-signal()

Advantage: We can do certain activities before other activities.  
eg Producer, consumer with buffer of 5.  
Only want Producer to produce 5, then wait.  
So set S=5, so produces 5, then must wait for consumer to deplete buffer.

Synchronise the order that processes occur in.

**Semaphore with no busy waiting (gets around busy – waiting resource wastage)**

Keep track of number of processes associated with struct. (list)

-block()  
-wakeup()

Processes waiting (blocked) go to sleep, until you wake up a process to go into critical section.

**Deadlock and Starvation**

Deadlock: 2+ processes waiting indefinitely for something (event) one of the waiting processes can do.

Inverse of wait and signal ordering must occur:  
wait(s)  
wait(q)  
…  
signal(q)  
signal(s)  
  
If you did q and s the other way around, a deadlock can occur.

Starvation (Indefinite blocking): A process may never be removed from semaphore queue which its suspended in (infinite sleep)

Priority Inversion – Scheduling problem when lower -priority process holds lock needed by higher priority process.  
  
eg   
P1 – highest priority  
P2 – medium priority  
P3 - Low priority  
  
P3 starts, now running  
P1 starts, but is blocked, waiting for P3  
P2 starts, but runs and takes up all CPU time (as higher priority than P3). Meaning P3 never finishes, so P1 never runs.

-Solve this by temporarily increasing priority, or (and?) using **priority inheritance protocol.**

**Problems with semaphores**

Incorrect use of semaphore:  
-Signal(s)…wait(s)  
-wait(s)…wait(s)

**Monitor**

High level abstraction where many things such as waits, signal ect are all done for you behind the scenes.

**Alternative Approaches**

Transactional Memory: Keeps a transaction of each thing done in buffer memory, only once task is done does it write/complete transaction. Any errors means it doesn’t write and can roll back

OpenMP: To protect a process in the critical section from being interupted

Functional Programming Languages: Maths language

**Deadlocks**

Challenge of holding but not releasing resource.  
Solution: all processes release all resources (restarting process), the hopefully enough resources to finish

**System Model**

Each process utilises resource as such:  
**Request** resource  
**Use** resource  
**Release** resource

**Deadlock Characterisation (NB theory questions asked)**

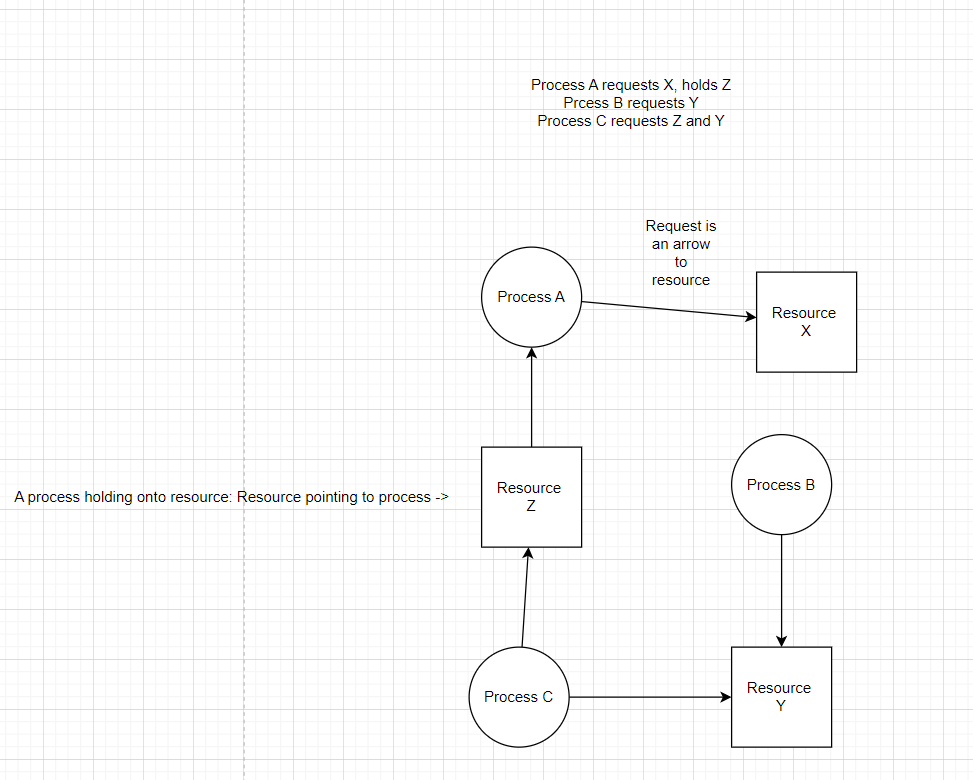
**Mutual exclusion:** Only one process at a time can use a resource

**Hold and wait:** Process holding one resource, is waiting for another resource to become available

**No pre-emption**: No 3rd party that can end a deadlock

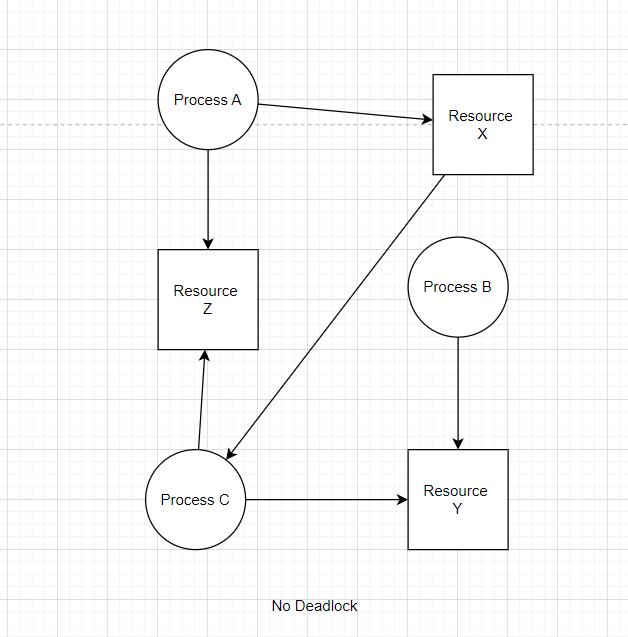
**Circular Wait**: P0 waiting for resource P1 has, p1 waiting for p2… p waiting for p1

**Resource Allocation Graph**

****

**A diagram of a process

Description automatically generated**



**Basic Facts**

No cycle in graph = no deadlock  
Cycle = possible deadlock (if more than one of a resource type or not)

**Methods for Handling Deadlocks**

Don’t have all 4 deadlock characteristics.

All processes let go of resources and try again.

Ignore problem (actually common)  
-Building systems to stop/detect such, is often more resource intensive