- Mutual Exclusion / Critical Section

- Synchronization

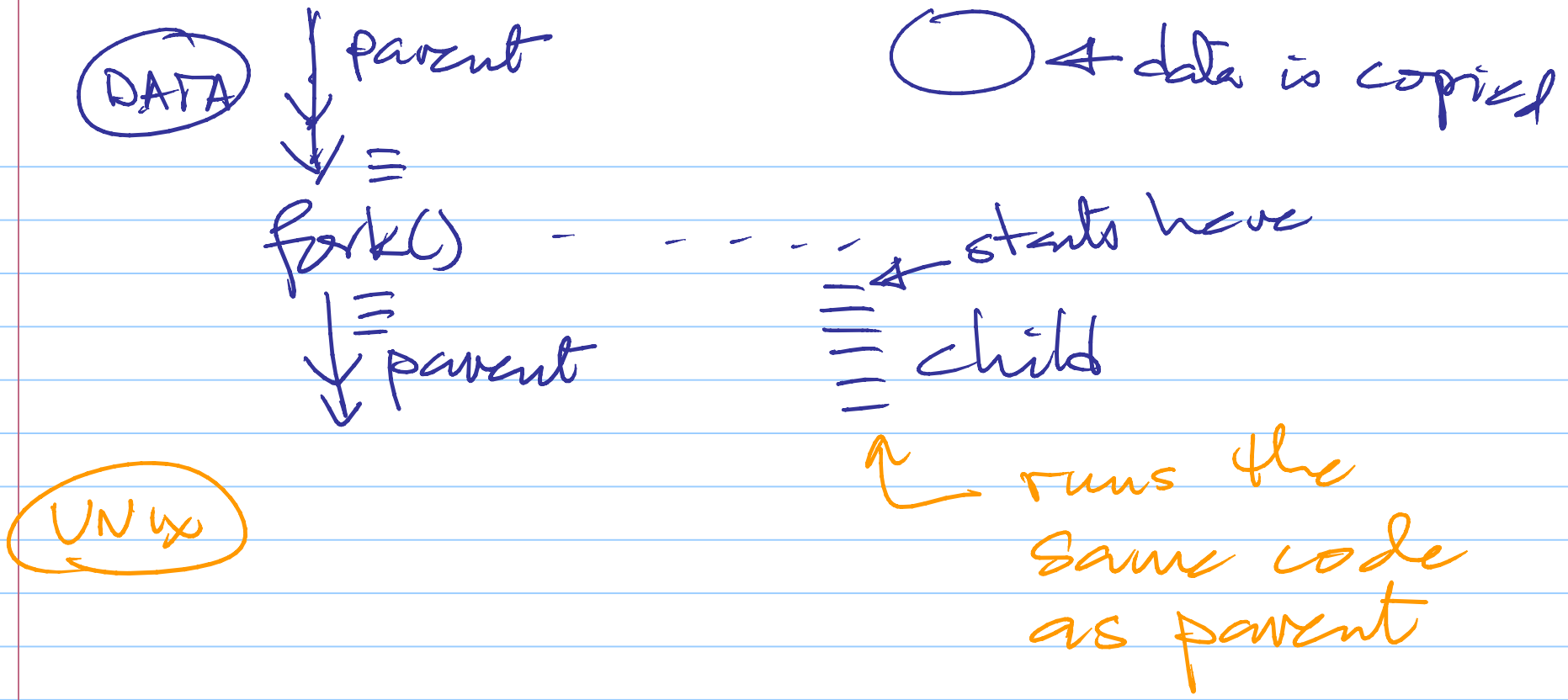
Shared memory
- Threads (App)
- Threads & processes (Kernel)

How to start a process

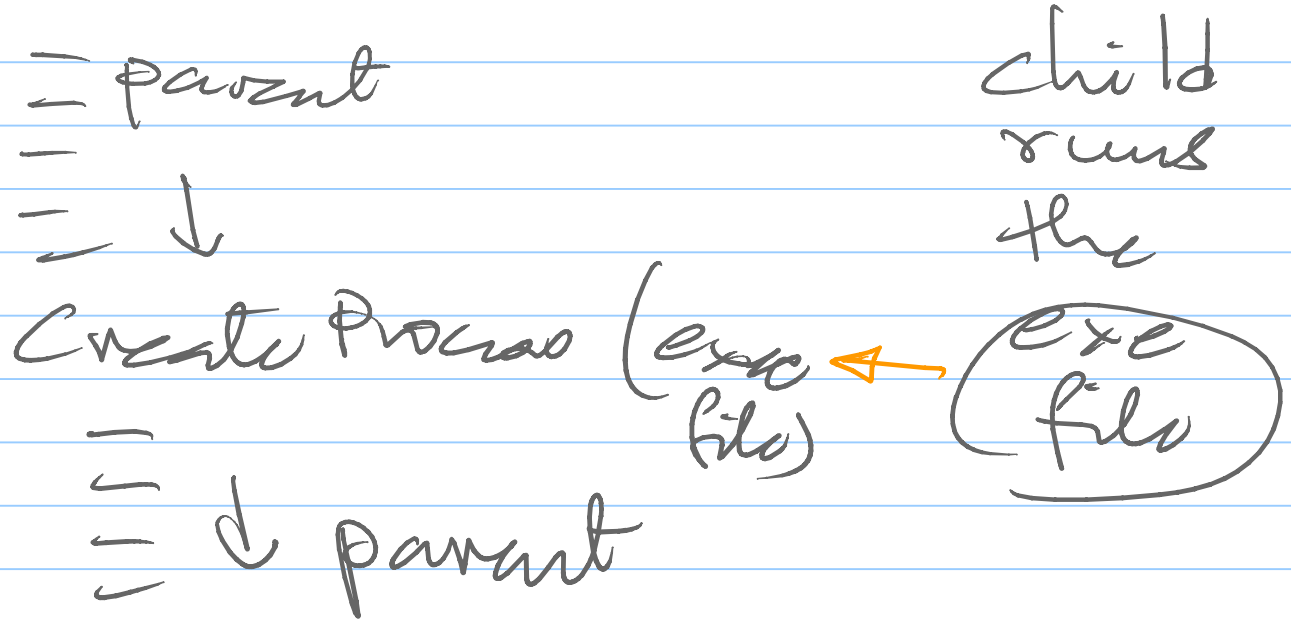
- A process starts a process
  - system call
    - UNIX/Linux
      - fork() + exec
    - windows
      - CreateProcess(arg)

- new PCB
- new address space

↑ privileged actions


DATA
parent
fork()
parent
child
starts here
data is copied
runs the same code as parent
UNIX

Windows


parent
CreateProcess (exec file)
child runs the exe file
parent

Thread Create

→ UNIX → none

LINUX → clone()

↳ just like unix fork() but shares memory

Windows CreateThread(arg) ↑ funct

- Do not use!
- Use pthreads → which uses clone.

parent

CreateThread(function)

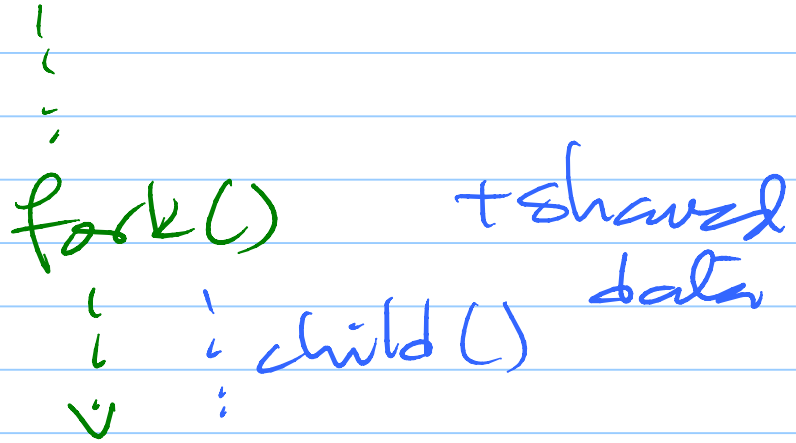
parent

child runs function

pthread_create(function)

→ same as windows

Threads uses fork-join (theory)



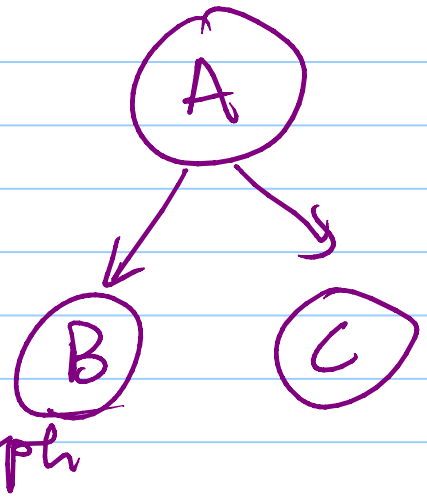
A stack variable

```
x = fork() // x will be a positive # in parent & 0 in child
```

B

```
if(x==0) { child code }
else { parent code }
```

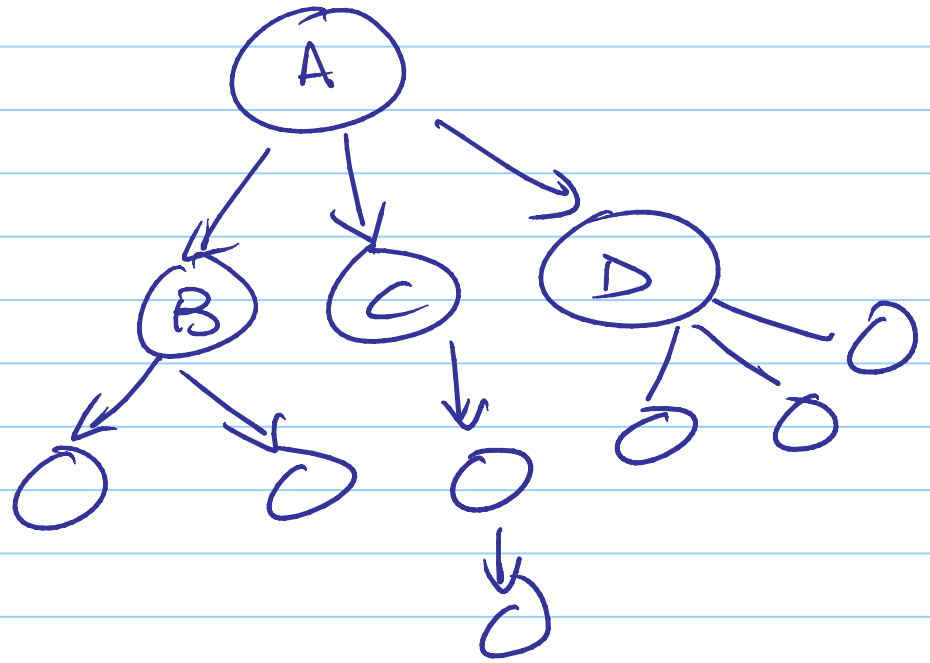
C



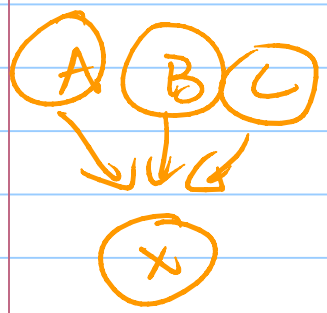
precedence graph

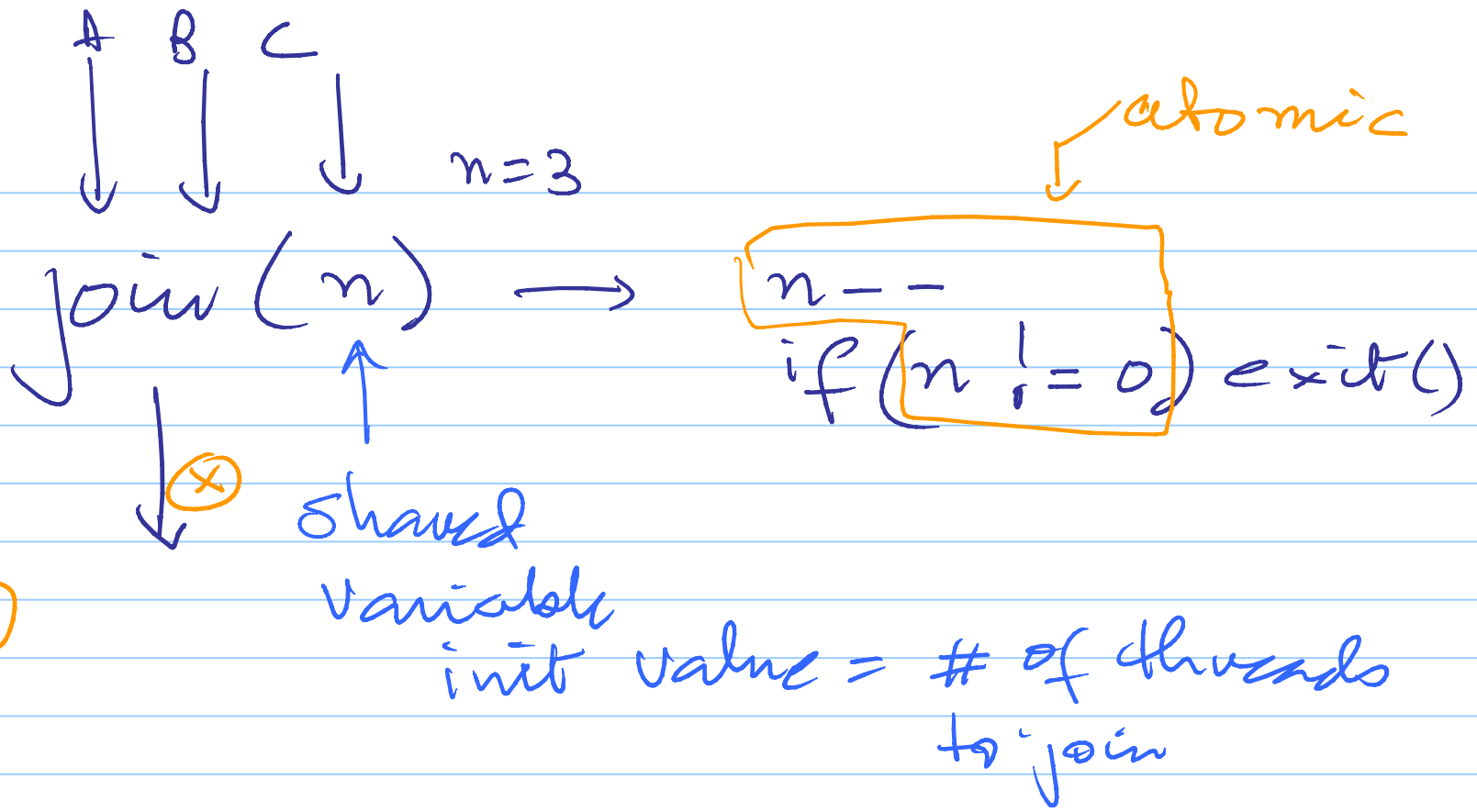
precedence graphs

fork →



fork
join
Join
A
fork
Join
Join


A
B
C
X


A
B
C
n=3
join (n)
X
shared variable
init value = # of threads to join
atomic
n--
if (n != 0) exit()

fork-join ⇒ can do any precedence graph

↳ can be used to implement all locking/sync situations

→ unstructured

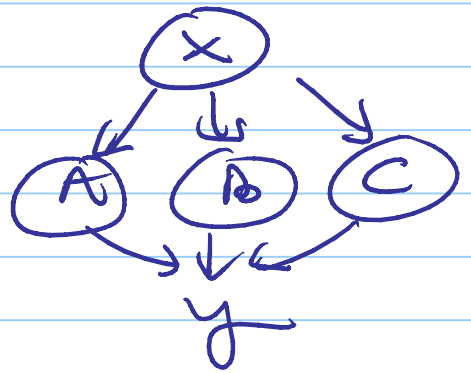
The 'par' construct

x

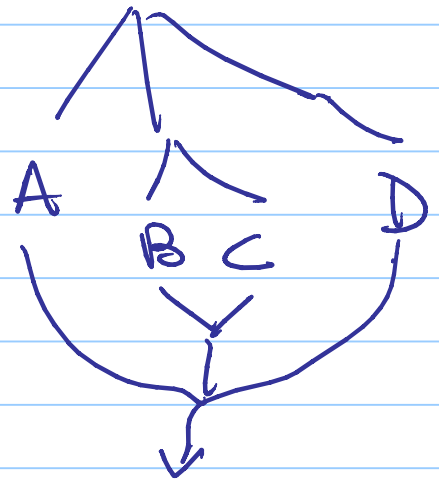
par { A ; B ; C }

y (set of statements

graph →



par { A; par { B; C }; D }



```
parfor (i = 1 => n) { ... code ... }
```

↑ iteration loop

- iteration loop runs first & generates n values for i
  - the code runs with n threads with i initialized to the n value

par / parfor ⇒ openMP // high level parallel programming

---

low level

- pthreads + ~~semaphore~~ or monitor

pthreads ...

pthread_create(f, ...)

pthread_exit()

pthread_wait()

↑ start a thread executing in f

Critical sections → locks (mutex)

pthread_mutex_t mutex_v // declare

→ pthread_mutex_init (mutex_v)

↳ set lock to 1

init

pthread_mutex_lock (mutex_v)

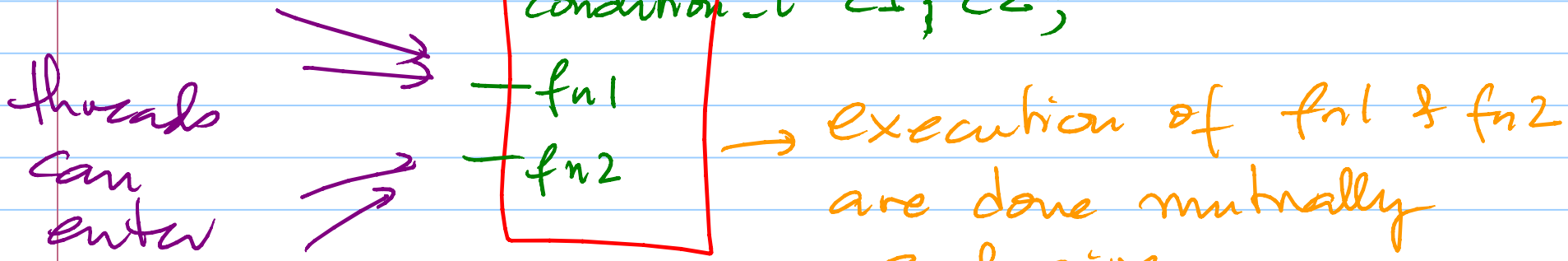
" " unlock (")

Synchronization → monitors

not pthread

abstract

monitors (class / object)



Monitor → methods + conditions

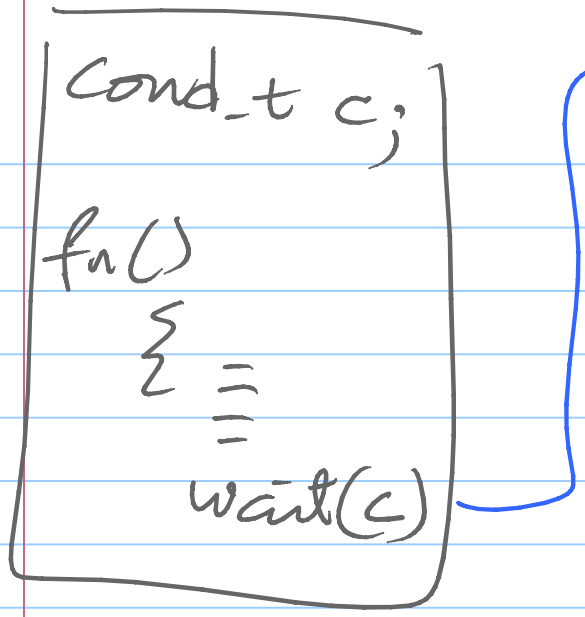
conditions: variable

↑ does not have a value

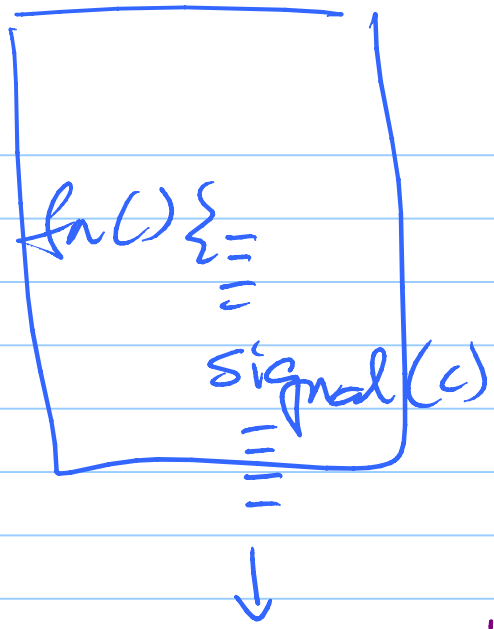
but supports functions

① wait (c)

② signal (c)



→ thread will get blocked. Always
- wait on a condition is an unconditional block
- another thread can enter monitor



Signal(c)

↳ wake up 1 thread that is waiting on c

→ many threads may be waiting

Woken thread has to wait for the signalling thread to (exit or wait)

→ no thread may be waiting

↳ NO_OP!

No memory!

f1()
wait(c)
f2()
signal(c)