CSE 330: Operating Systems

Class: 16 Date: 10/13

Fall 2016

Note Title

pthreads – threads for Linux

↳ start, stop, $\boxed{\text{sync}}$, mutex

sync → monitor

mutex ← binary semaphore

returns thread_id

pthread_create (&thread_id, NULL,
func, arg)

run thread in funct

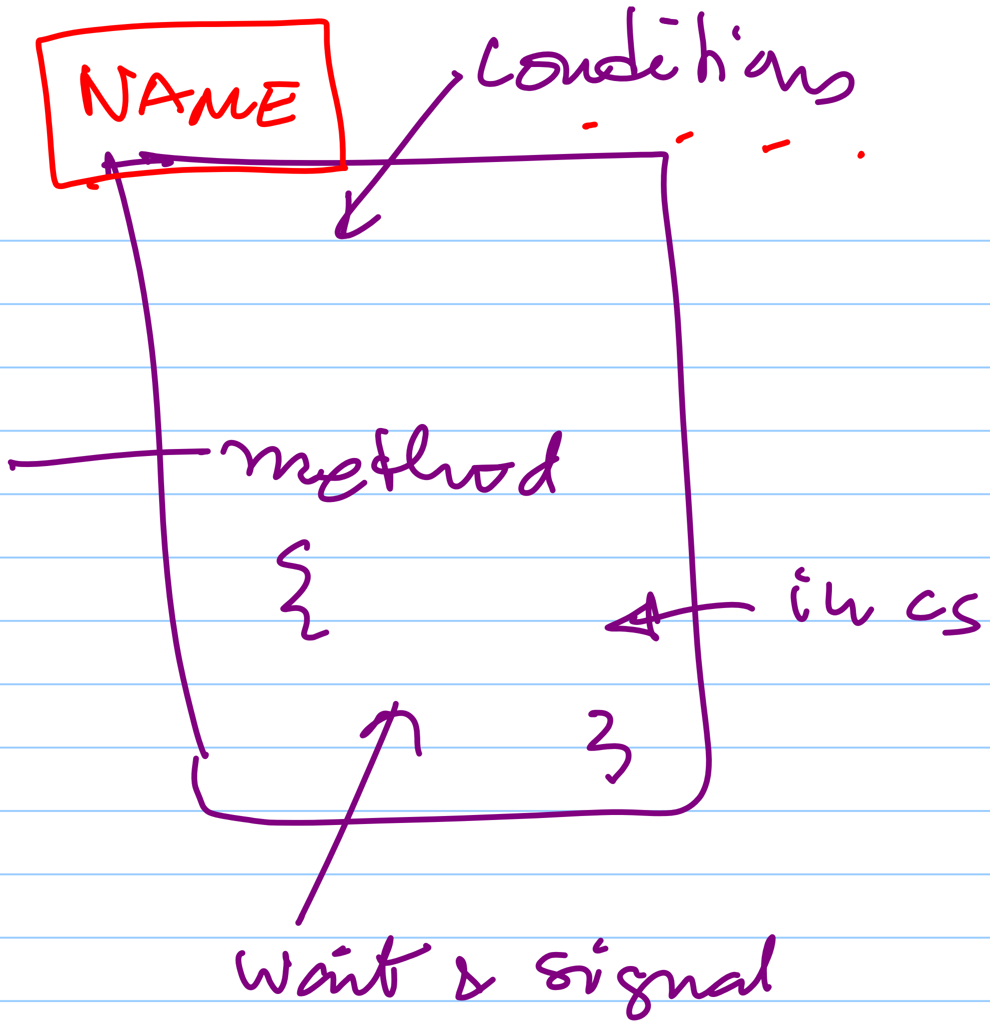
args

pthread_mutex_init (mutex)

pthread_mutex_lock (mutex)

"   "   _unlock (mutex)

pthread_mutex_t mutex;


NAME
conditions
method
{
}
in cs
wait & signal

monitor

```
pthread_mutex_t  mon;
pthread_cond_t  c1, c2;
```

conditions

---

initialize the mutex

" " conditions

```
pthread_cond_init (c1)
  "     "     "   (c2)
```

f1 (arg) → is a method in monitor
mon

```
{ pthread_mutex_lock (mon)

      wait  ————————→

      signal  ————————→

  pthread_mutex_unlock (mon)
} ← pthread_yield()
```

wait(c1);

↳ pthread_cond_wait(c1, mon)

condition

the monitor associated with c1

signal(c1)

→ pthread_cond_signal(c1)

pthread_cond_wait (c1, mon)

→ ⎡ unlock(mon)
  ⎢ put thread in Q inside c1; //block
  ⎣ lock(mon)

pthread_cond_signal (c1)

⇒ wakeup 1 thread blocked on
c1 if any

Implementing Semaphores.

Sem → struct [count, queue]

```
Init(Sem, v)
    → Sem->count = v;

P(Sem) { decr sem counter
         if counter < 0 block }

V(Sem) { incr sem count;
         if count <= 0 wakeup }
```

block

```
{   prev = Curr_thread
    AddQ( Sem queue, prev)
    Curr thread = DelQ (ReadyQ)
    swap ( prev -> context,
                  Curr_thread_context)
}
```

```
unblock
    { AddQ(Ready_Q, DelQ($\overline{\text{sem Q}}$))
      yield()
    }
```

```
f1() {
    ∞loop
        [ global++ ]
        yield()
    }

f2() { ∞loop
        global++ ]
        yield()
    }
```

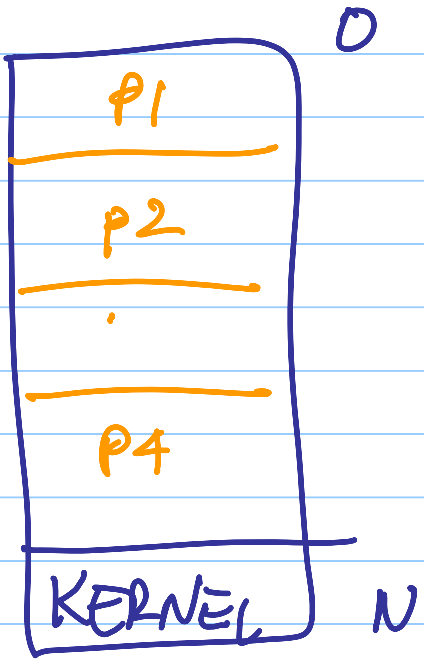
```
∞ loop
        P(mutex)
        global++
        V(mutex)      <-- busy the yield here

  ∞ loop
        P(mutex)
        global ++
        V(mutex)      <--
```

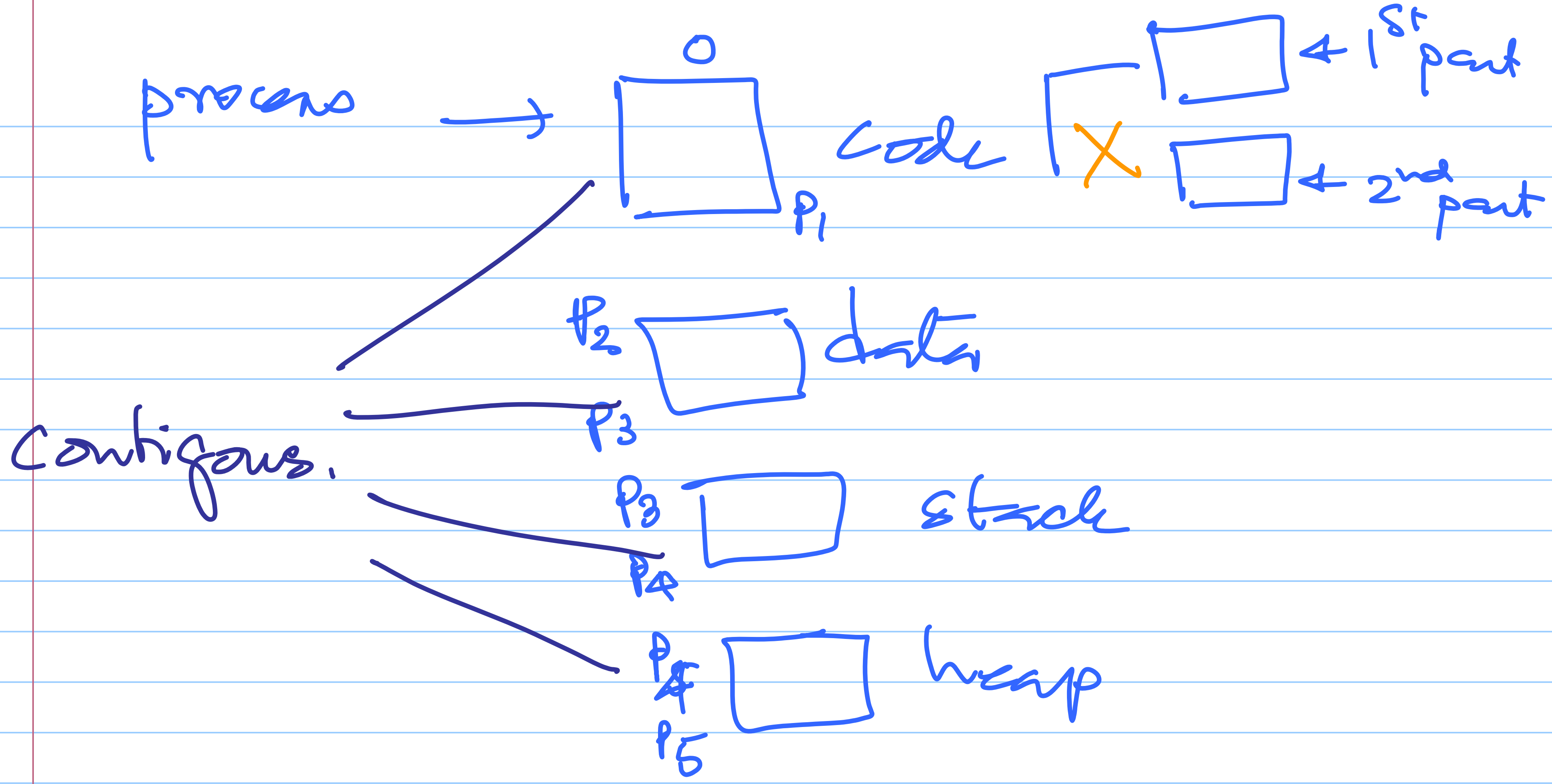
Memory management

physical memory mngmnt

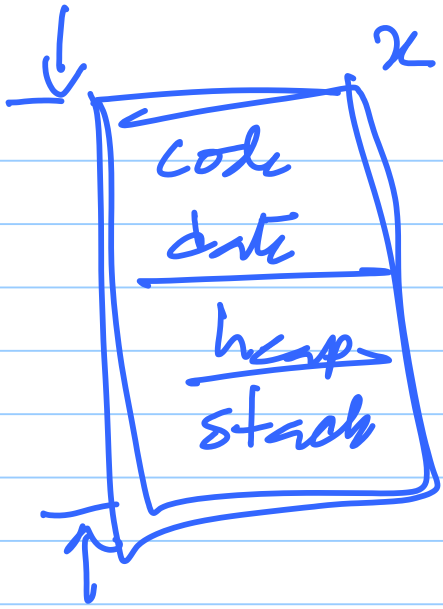
- get-memory & run processes

- dispose of mem @ termination

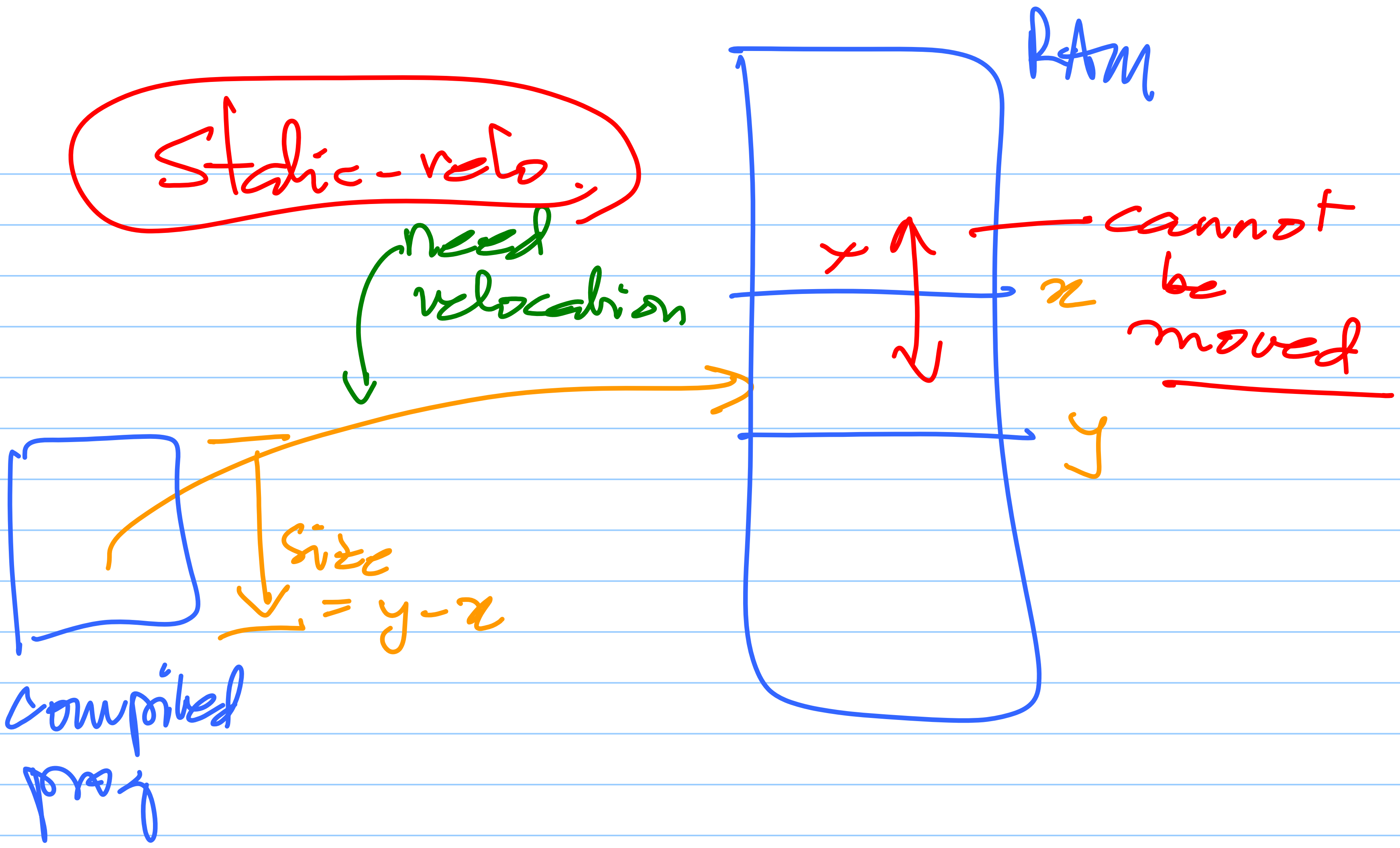


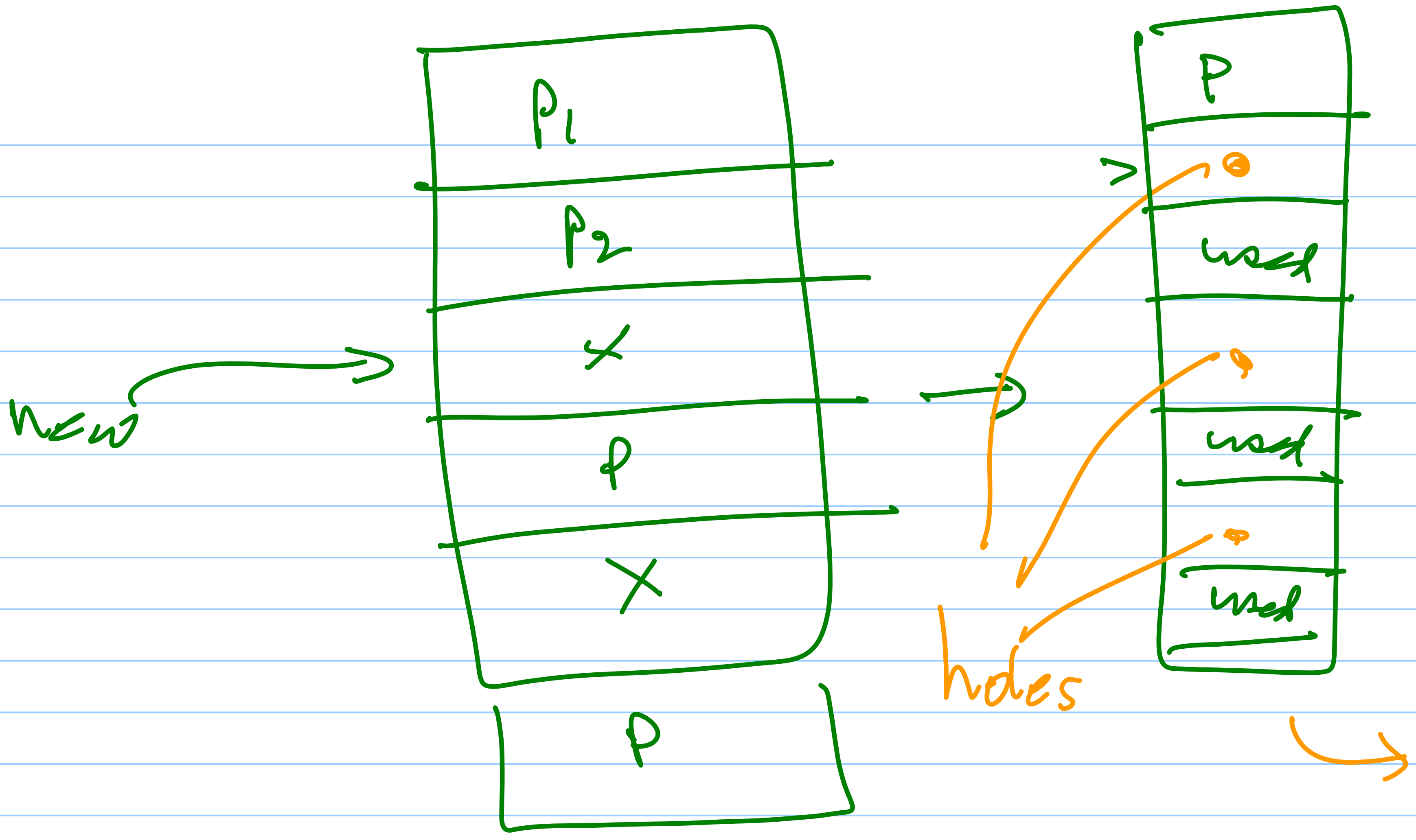

process
0
code
P_1
1st part
2nd part
P_2
data
P_3
Contigous.
P_3
stack
P_4
~~P_4~~
P_5
heap

process



one contig
memory range


Static-relo
need relocation
RAM
cannot be moved
x
y
size = y-x
compiled prog

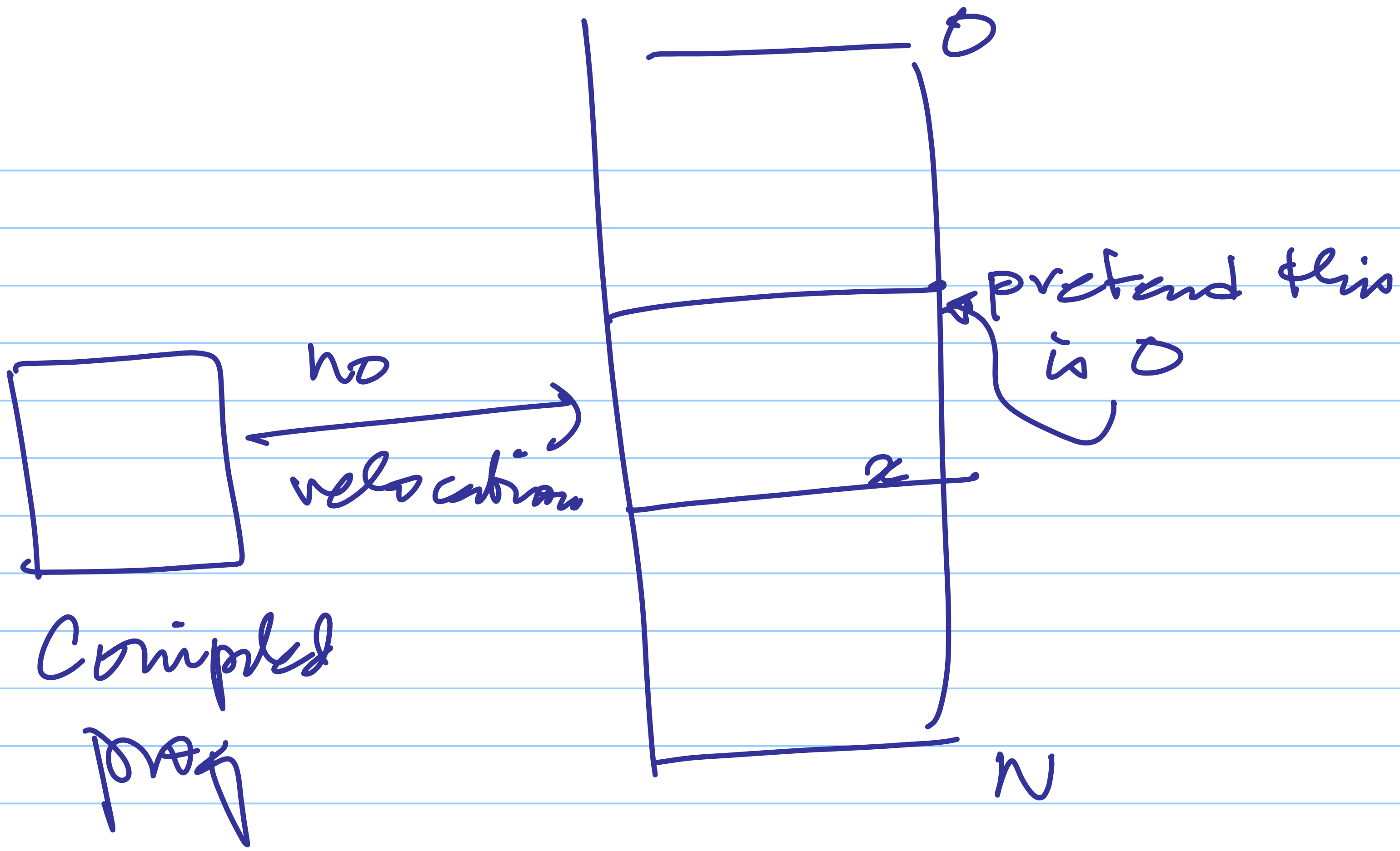

new
P1
P2
x
P
X
P
P
used
used
used
holes

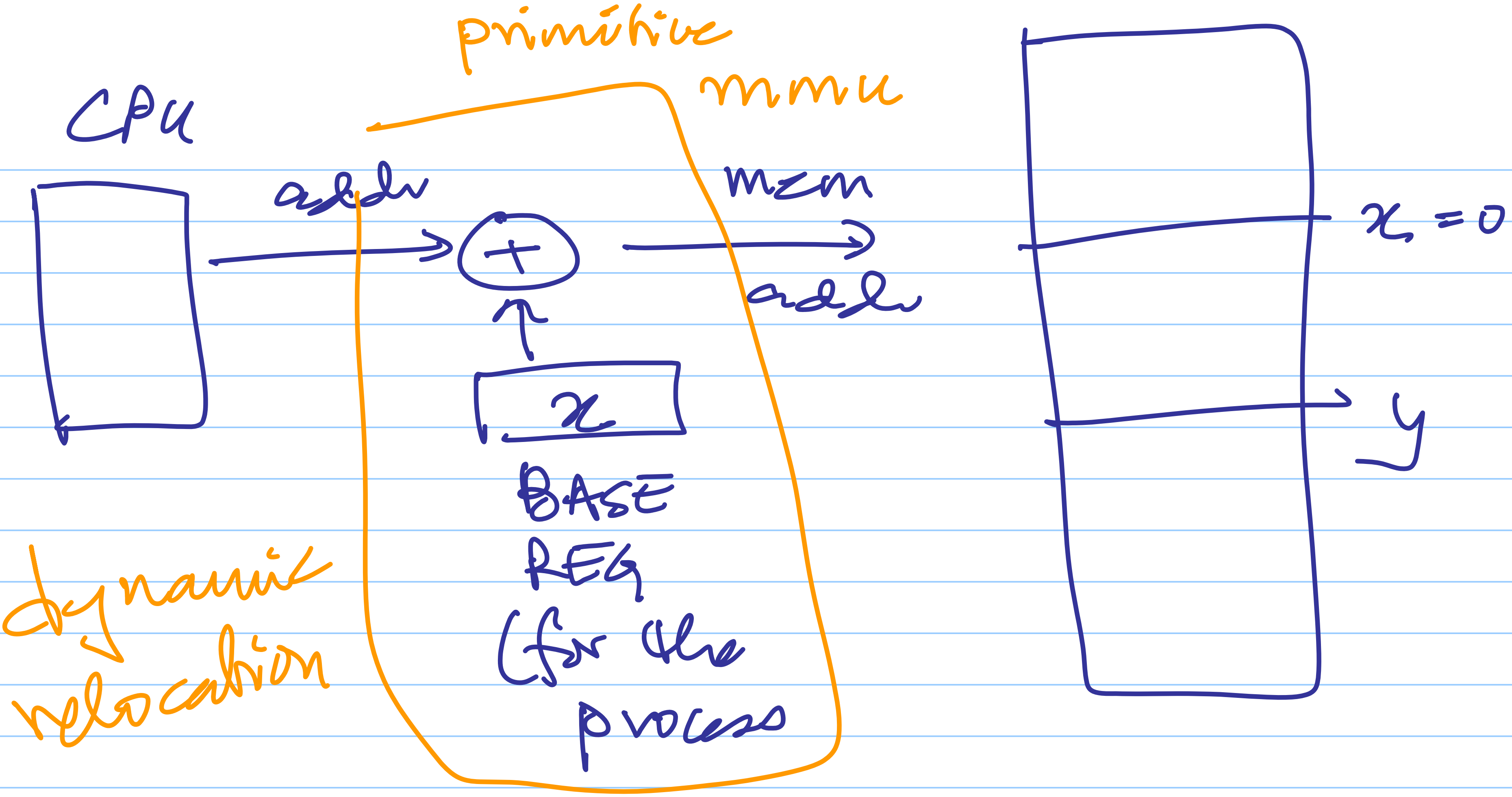
Fragmentation

- external fragmentation

  ↳ unused memory that cannot be used

  (too small)


no
relocation
Compiled
prog
0
2
pretend this
is 0
N


primitive
mmu
CPU
addr
+
mem
addr
x
BASE
REG
(for the
process
x=0
y
dynamic
relocation

Dynamic Relo

→ also has external frag

→ But can be fixed by

$\boxed{\text{compaction}}$

→ stop all execution → copy

→ coalesce holes

Overlays $\rightarrow$ use allocated mem of processes to add new modules / overwrite

Swapping $\rightarrow$ write the process mem to disk & then reload later