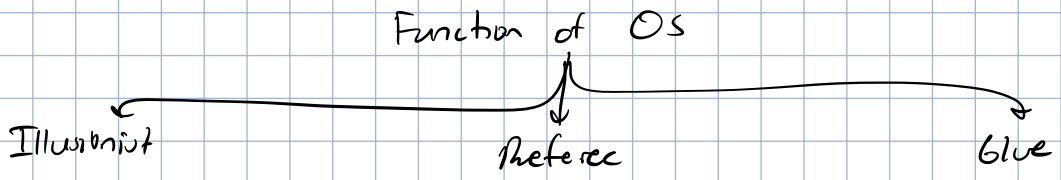
Introduction

What is an OS?

- What is shipped w/ device

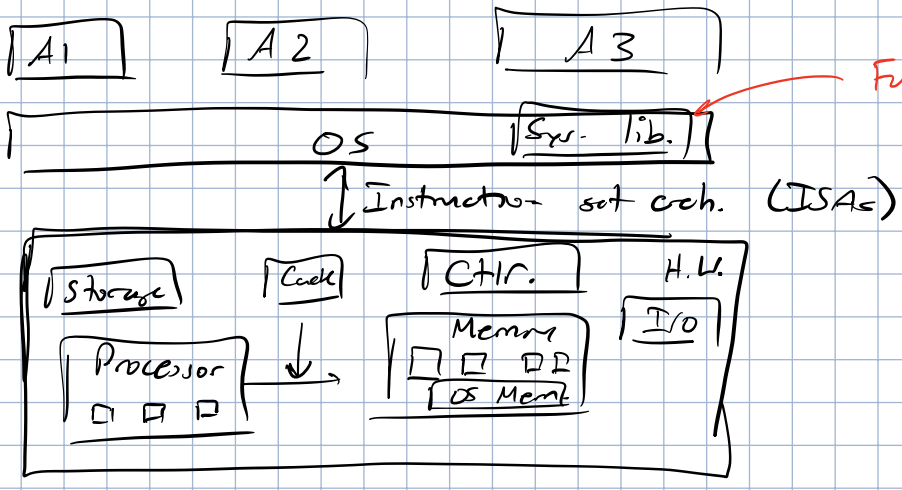
What does OS do?



Illusionist:

Clean abst. of H.W.

↳ No worries about resources, mem.

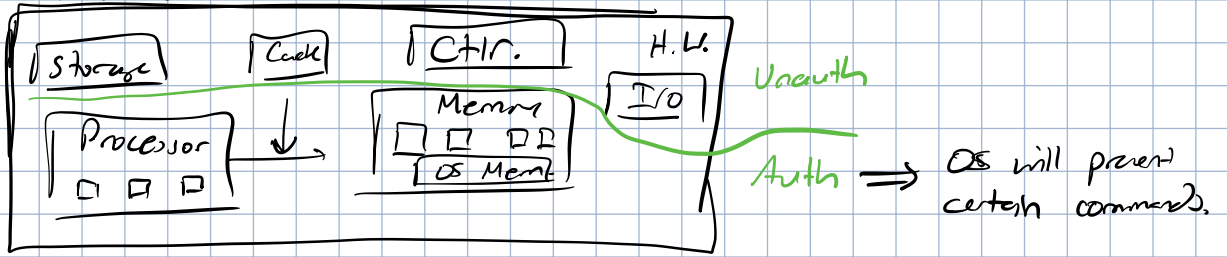


Examples of abst.:

- Processor → threads
- Memory → address spaces / virtual mem.
- Storage → file system
- Networks → sockets

Referee:

1. Schedule resource allocations:
2. Apps cannot access unauthorized resources.
   - Unauth resources:



- Ex:// If accessed unauth. memory ⇒ seg. fault.
- Resources alloc. to 1 process cannot be accessed by another process

Glue:

Common interface for H.W. ⇒ pretty easy to "access"

OS is not resource intensive!

↳ How?

1. Pair up w/ H.W. (not necessary, but easier & more lightweight)
2. Algos & D.S.

Kernel: runs all time on computer

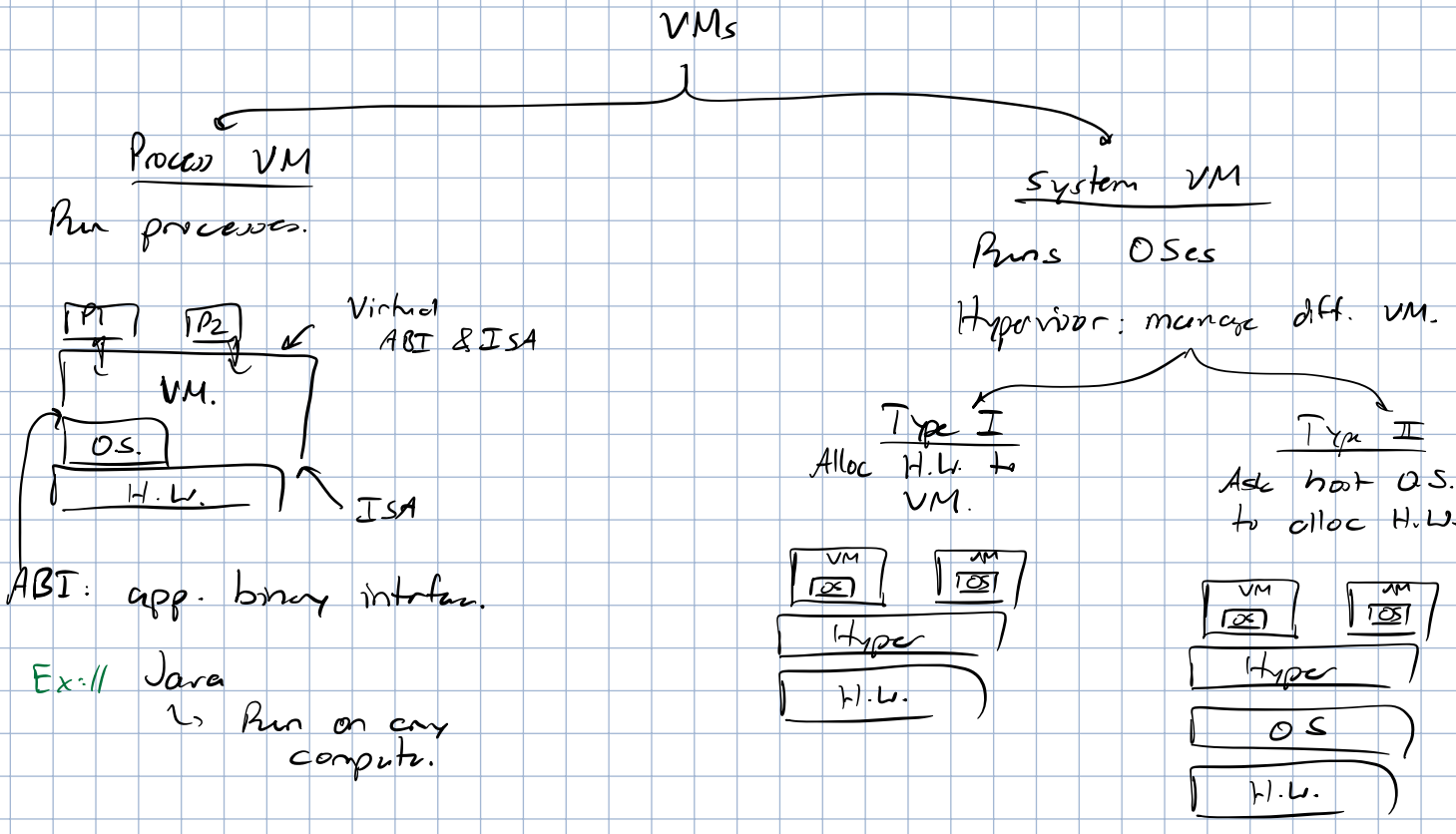
↳ Anything else is either an app or system app

Virtual Machines

Defn: software that emulates H.W.

↳ Doesn't depend on H.W.

ILLUSION ⇒ isolated H.W.



ABI: app. binary interface.

Ex:// Java

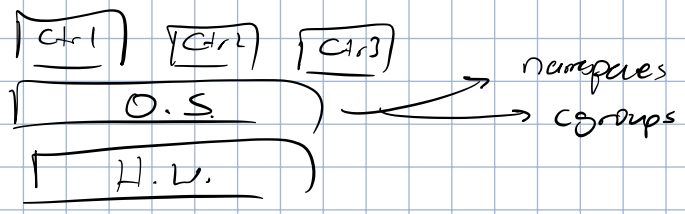
↳ Run on any computer.

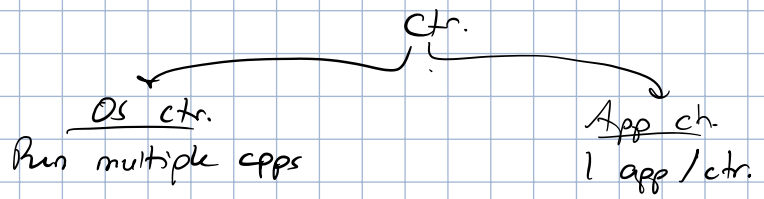
Container:

Defn: emulate O.S.

ILLUSION → isolated O.S.

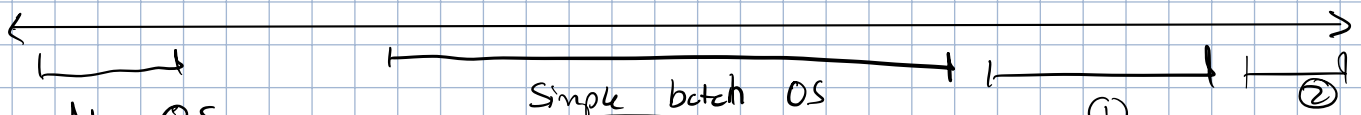
B/c only emulating OS ⇒ perf. benefits.





OS CONCEPTS

Brief History



No OS

- 1 user
- Cards = 1 prog ⇒ execute
- Manually link
- Pain

Simple batch OS

- Put cards in queue → machine would dequeue.
- Batch progs
- Serial ⇒ I/O slows down.

Multiprog batch OS

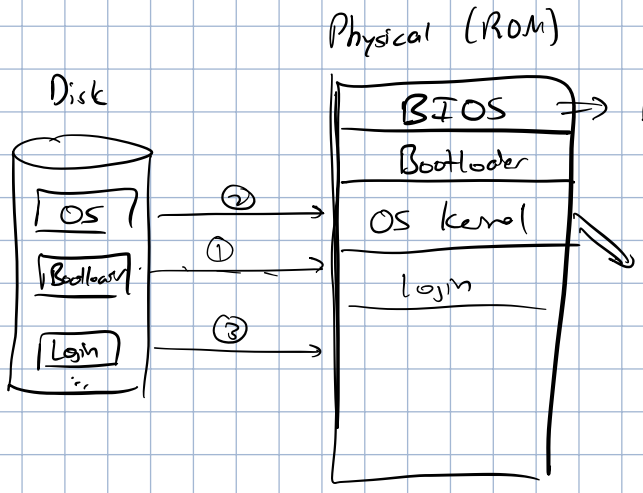
- If blocked ⇒ switch to other progs.
- CPU utilized, I/O is on the

Time-sharing OS

- Multiple users
- OS would use scheduler to allocate CPU time.

OS ← other OS

Booting



① BIOS loads bootloader

② Bootloader load OS

③ OS kernel → load login / shell

Q: Why do OS load last?

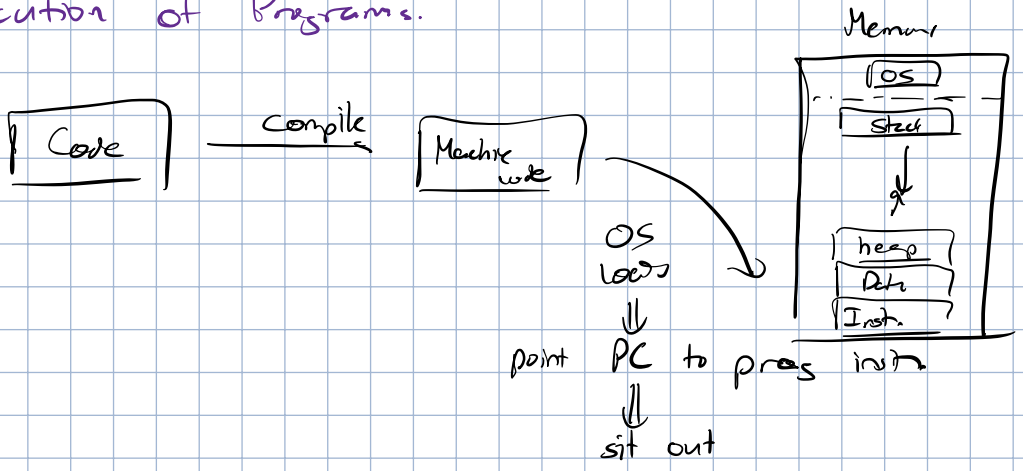
a) ROM is expensive ⇒ kernel is big

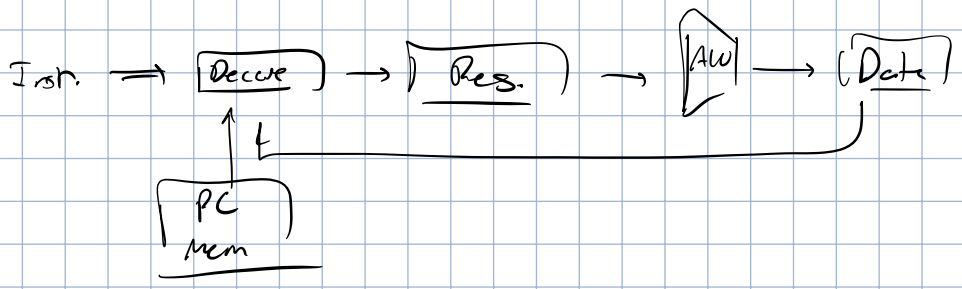
b) Kernel is dynamic (update) (hard to update if in ROM)

Q: Why bootloader b/c OS?

a) Choice on OS

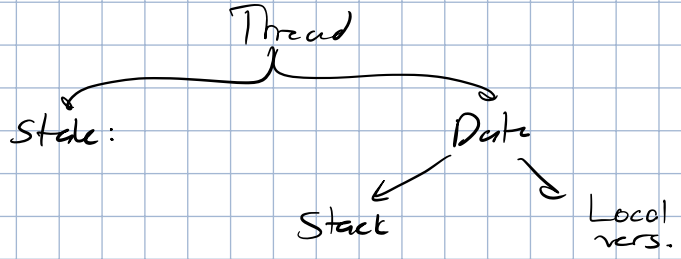
Execution of Programs.





Thread

Defn: Seq. of commands on CPU



Multithreaded: 1 prog using multiple threads?

↳ CPU can only 1 thing / time. ⇒ Can do more w/ more threads. } thread creates new thread # parent + child

↳ Can use concurrency to illusion of parallelism

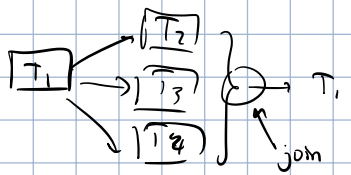
pthread: linux lib.

Commands

pthread_create

pthread_exit
1. Cleanup + quit
2. Lock up waiting threads
3. Child thread continue (unlike exit())

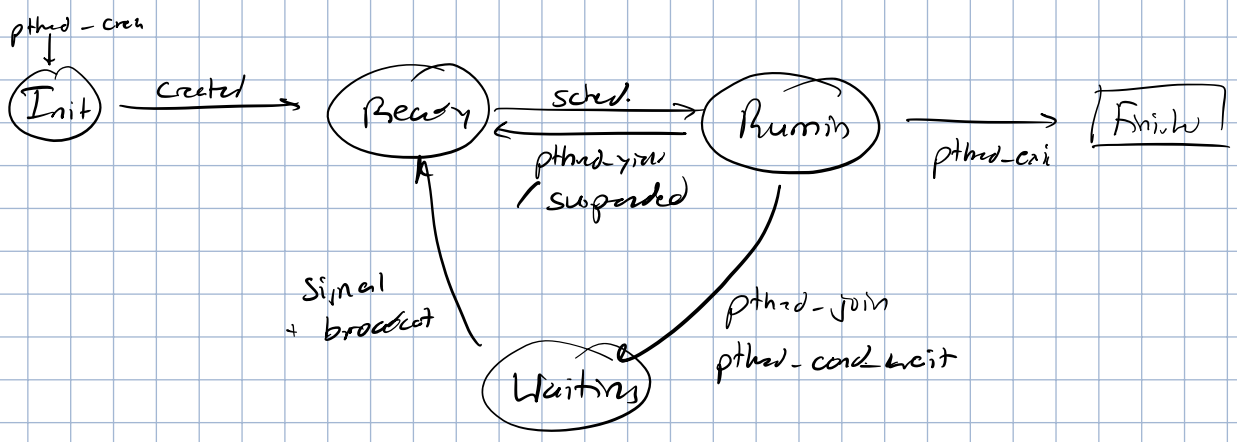
pthread_join



pthread_yield

Relinquish CPU
⇓
OS look at thread pri. ⇒ alloc

Thread state:

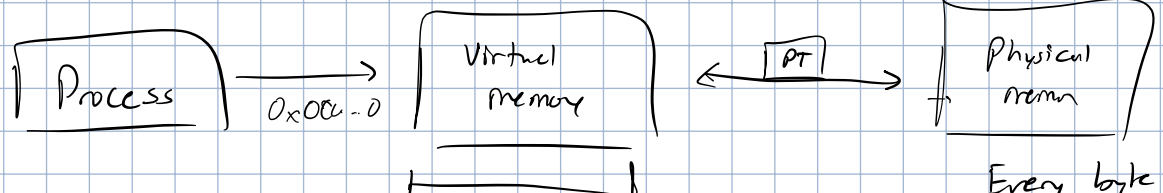


Thread control block (TCB):

Keeps metadata
- TID
- PC / memory ptr to stack
- Reg values.

} Need b/c concurrency ⇒ switch b/w threads, how do we restart? + context switch

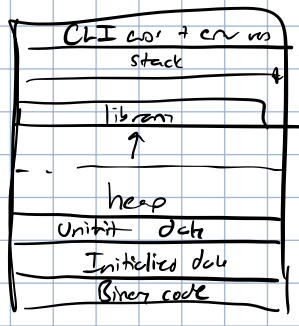
Address Space



Per program
If n bits ⇒ 2^n addr.

Every byte has an addr
∴ 4 GB ⇒ 2^{32} B of addrs

Memory organization in virt. mem:



← Scoped variables.

} Uninit space

Libraries are not fully imported
↳ Dynamic linking, full import of func. only if used.

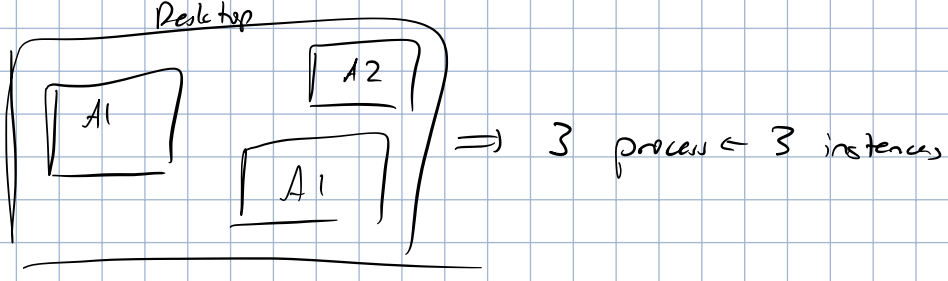
Stack:

↳ Var values (args are pushed b/f function jump)

↳ Return address

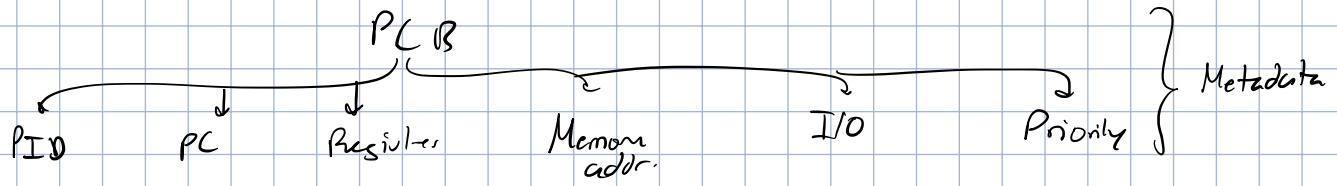
Process

Defn: prog. in execution



⇒ 3 process ← 3 instances

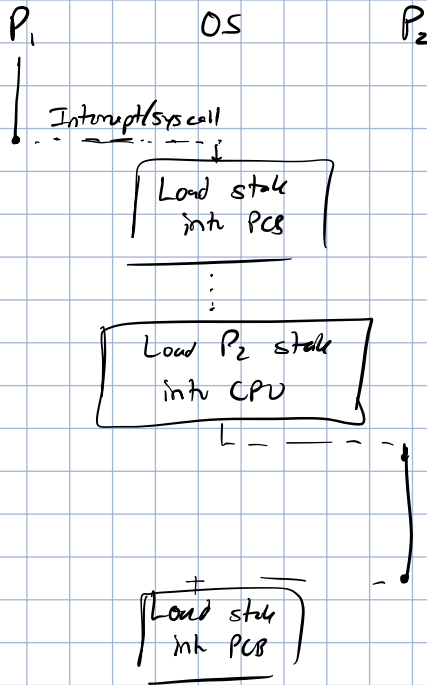
Every process has a process control block (PCB)



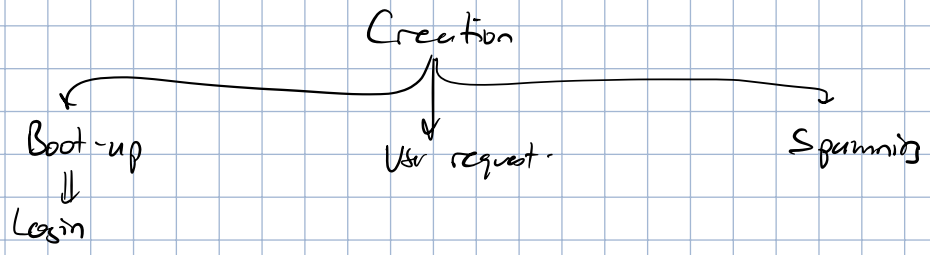
1) Process creation → PCB create → OS append to list of PCBs

2) Process exits → OS will pop off PCB

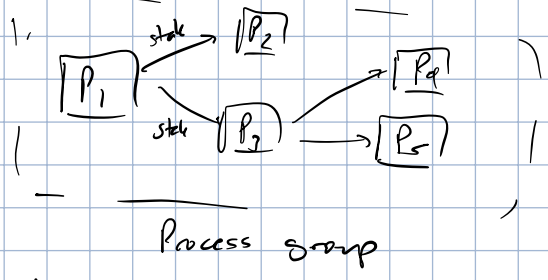
Process context switch: run multiple process on 1 CPU



Process creation:



Process spawning.

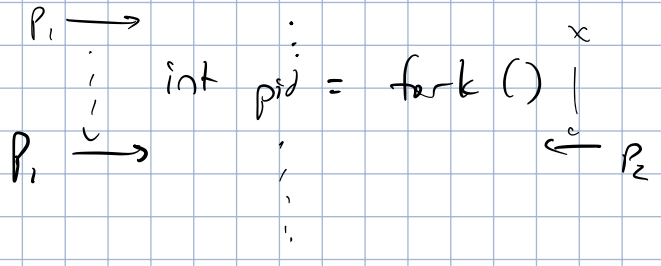


Why?

① Open up diff program to do smth.

② Organization

How?



if pid == 0 ⇒ parent process

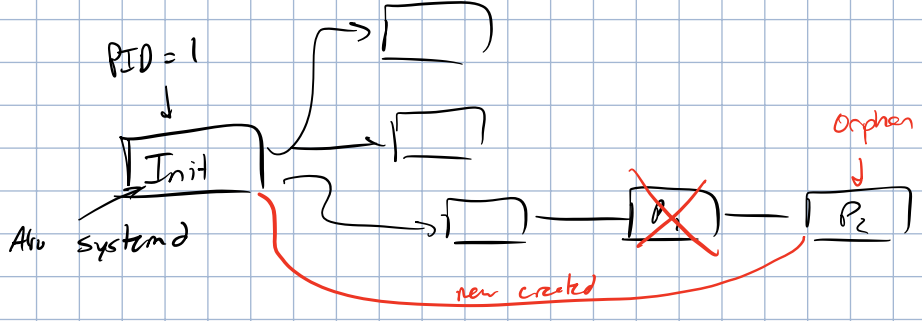
else if pid > 0 ⇒ PID of child proc.

To wait for process to finish: wait (pid)

Note: fork bomb

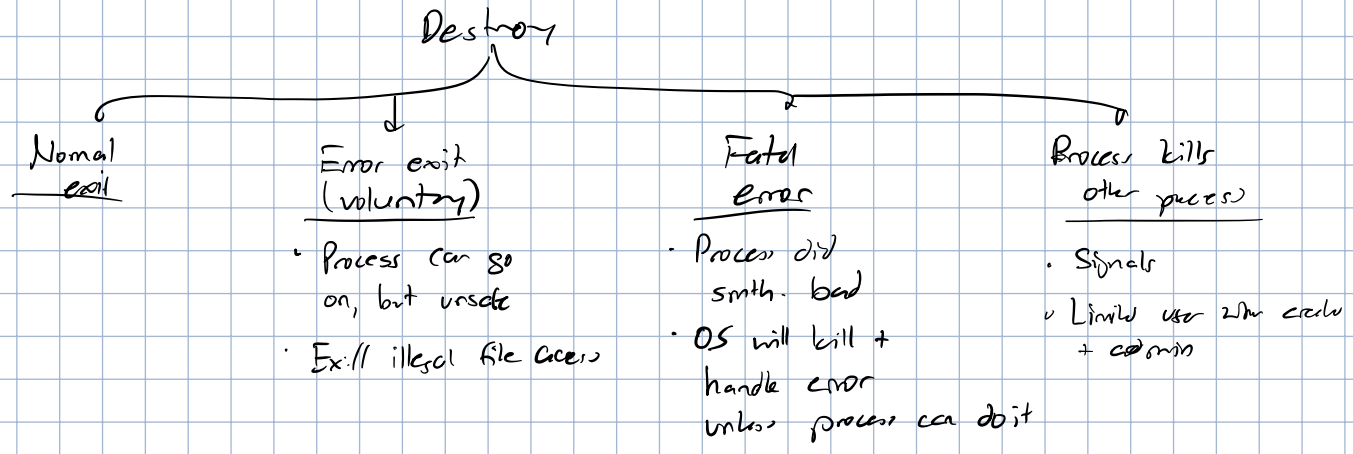
Ls Combat: limit # of process / rate of creation

Family Tree:

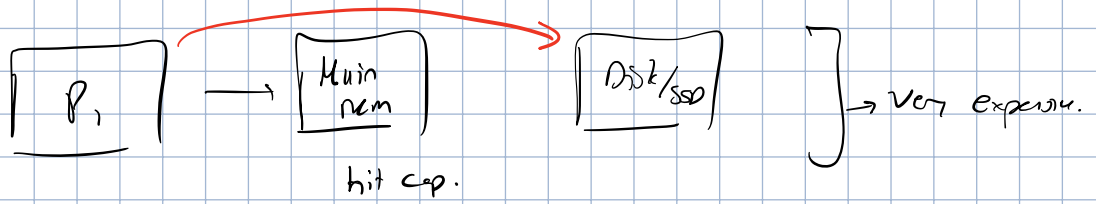


- Func. of init: kernel, pgirg

Destruction:

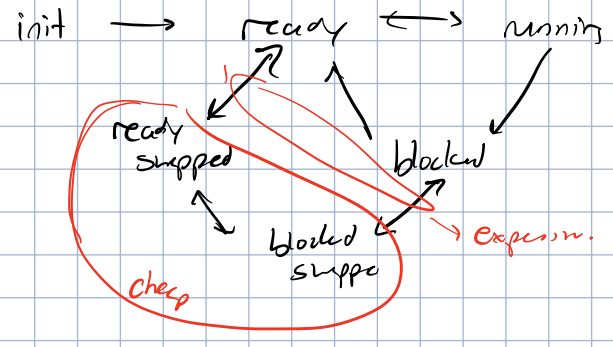


Run out virt. memory ⇒ request too much mem. ⇒ swapping.

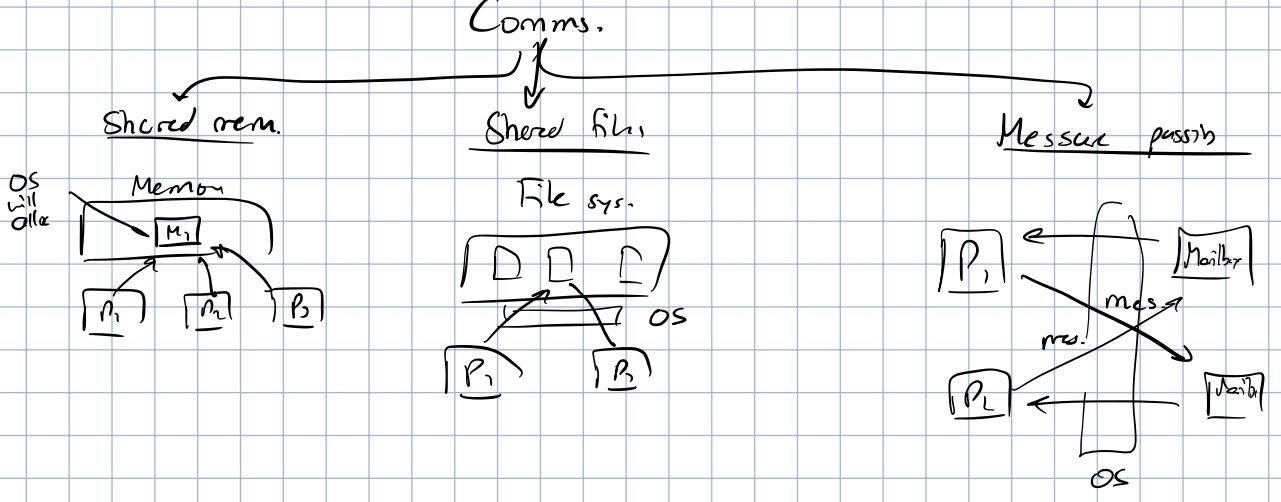


- Like: blocked
- Keep p table

State diagram



Inter-process comms.



Message passing:

① Direct: use process to indicate destn. / recip.

send (P_1 , message) recieve (Q , message)

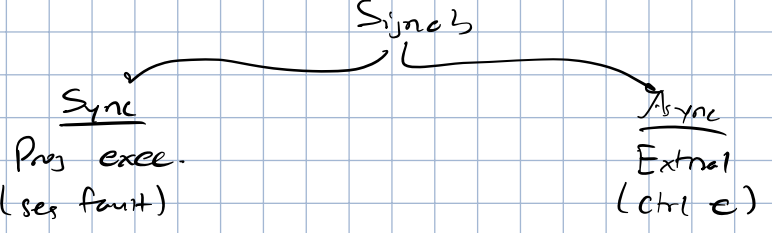
② Indirect: use proces mailbox

send (MB_1 , messa), recv (MB_1 , messa)

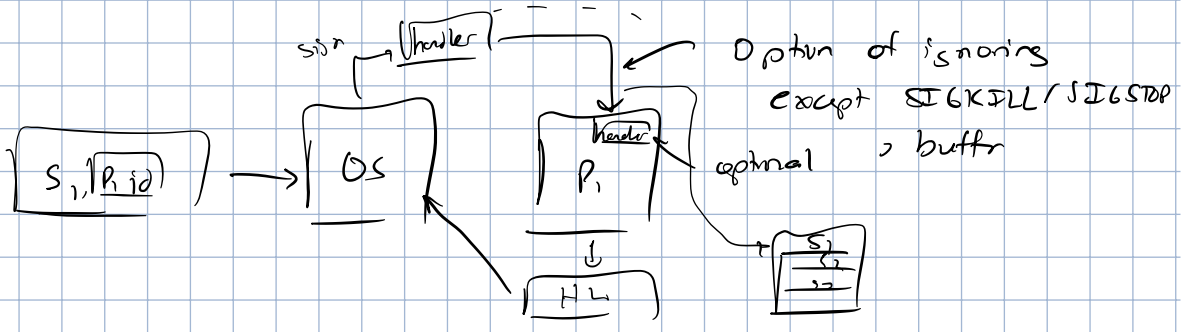
Message can be blocking / non-blocking

Signals: limited direct IPC

no data needs proces no.



Working:



Ex: // kill -9 PID

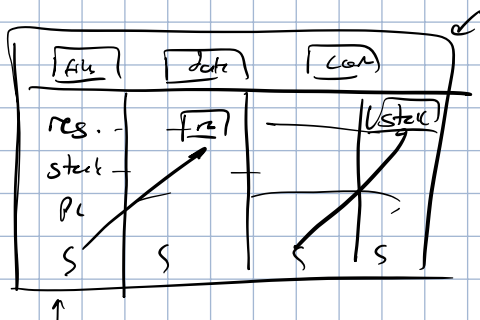
↳ SIGKILL signal

Custom handles should be pure ⇒ no side effects (malloc)

↳ reentrant functions

Processes vs. threads

Multithreaded proces:



Threads in not isolated w/in proc

↳ Comm. are really easy

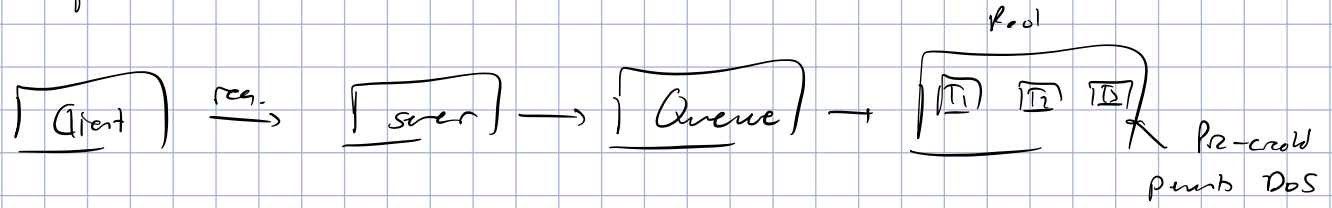
T1

CPU can only handle 1 thread / time ⇒ switch b/w thrds ⇒ Concurrency

Why multi threads vs. multiprocess.

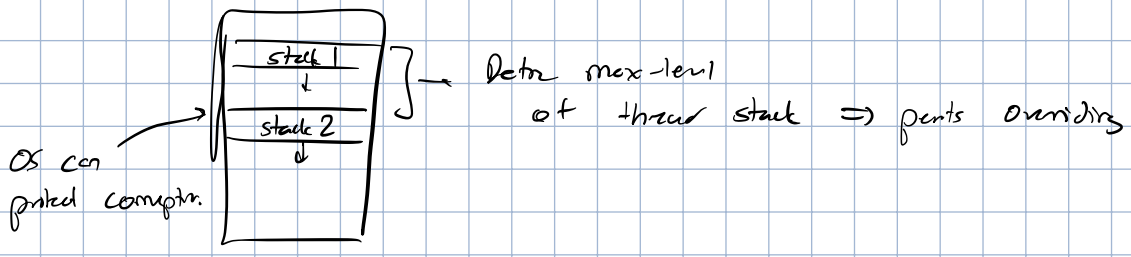
↳ Process are expensive to create, need OS

Thread pools for servs:

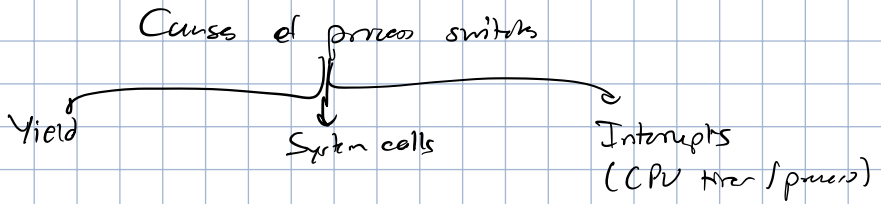


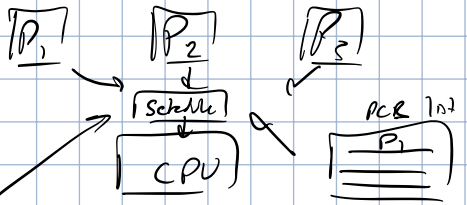
Tradeoff: speed vs. security

How to handle memory of multiple thrds in 1 VM:

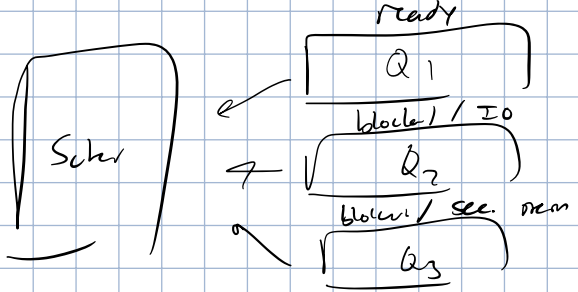


Multiprogramming



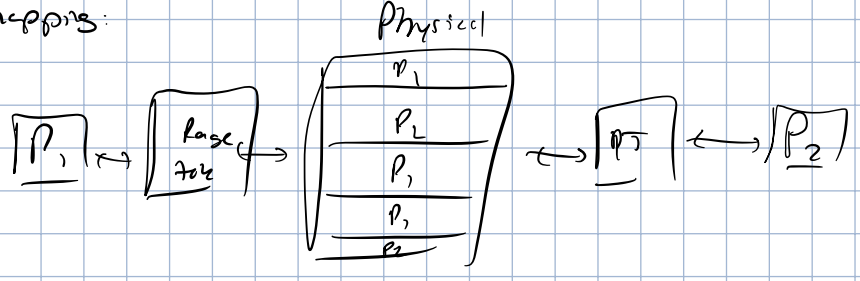


Zoom:

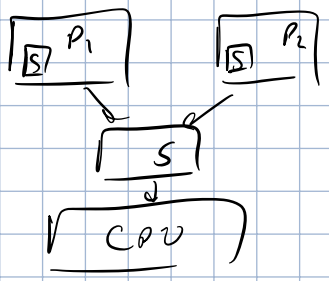


Q: multiple process switches → protect?

Memory mapping:



1st Version: basic unithread

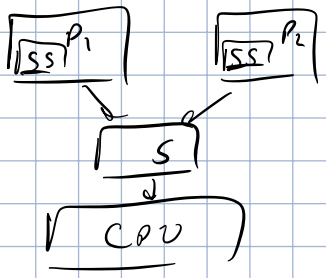


Switch overhead: hi

Protection: good

Sharing: not good

2nd ver: basic multithread

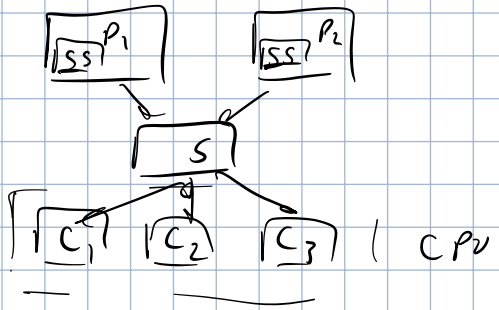


Switch: medium

Protection: medium

Sharing: medium

3rd ver: multicores

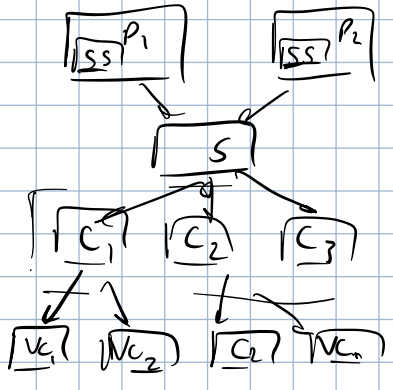


Switch: low

Protection: —

Sharing: low (thread simultaneously)

4th ver: hyperthreading:



Switch: extremely low

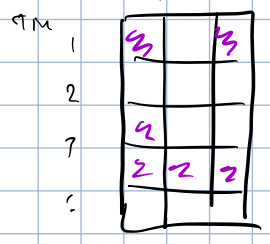
Protection: —

Sharing: —

Performance: ↓

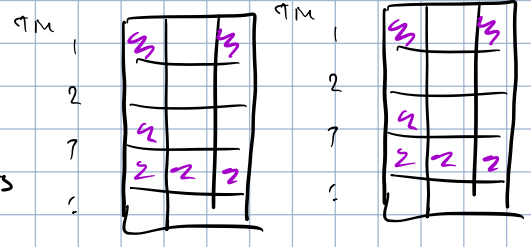
Evolution:

Superscalar

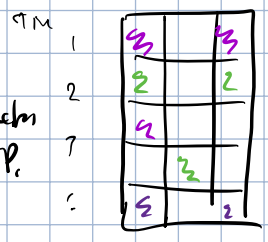


1 process ⇓ wait b/c blocking

Multiprocessor



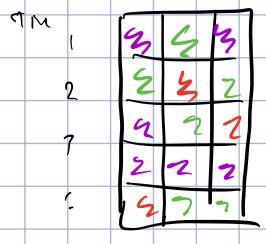
Fine-grain multithreading



Switch b/w P.

1 processor
3 ALU

Simultaneous multithreading / hyperthreading



⇒ Proces executing simu.

High utilization

Hardware is coordinated:
OS aware it has virt. (more processes to run)

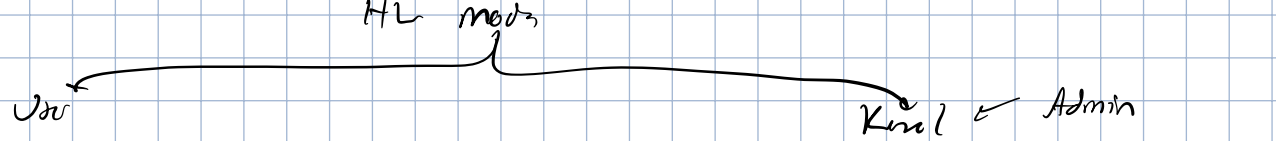
Tradeoff: performance vs. utilization

Only phys. cors ⇒ util ↓, perf (no thread comp.)

// virt. cors ⇒ util ↑, perf ↓ (more comp. → delay)

Sun: 2 VC / physical cr.

Dual-Mode Operation



Why?

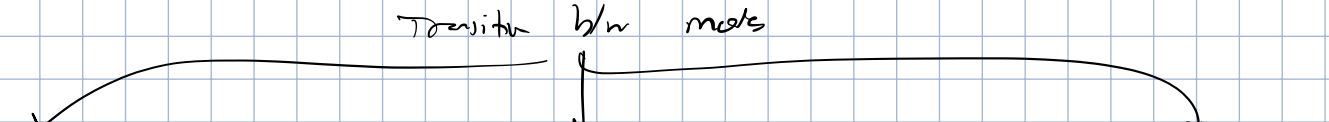
- User shouldn't have admin privileges. ⇒ uncontrollable

Any code in kernel mode = kernel

Not necessary to run kernel in OS ⇒ OS will need to interpret code

H.W. is needed

↳ Single bit for mode
↳ Transition b/w modes
↳ Prevent actions



System call

Requests kernel services

Requ → trap/syscall → kern

Proveor excpt.

Internal, sync. event from hardware

⇓

from code

Interrupts

External, asynch. evnt

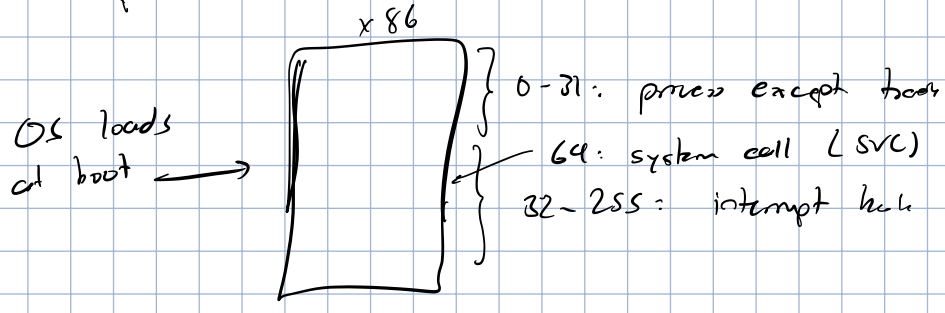
Can be disabled

Requirements for safe instr

1. Limited entry into kernel
   - APIs few
2. Atomic change in processor state
   - Chge: mode, SP, PC
3. Reversible
   - Easy state restoration after mode transfer.

Q: handling?

Interrupt vector table:

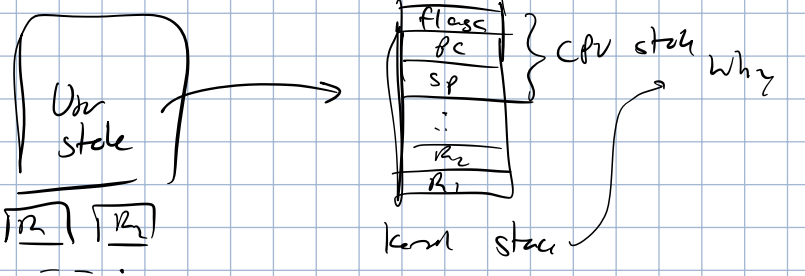


HW evnt → check IVT → load up handler core → exec.

Q: saving state?

State in use → kernel reusible

Use a kernel stack



Why not user stack?

Other threads can access + change

⇓

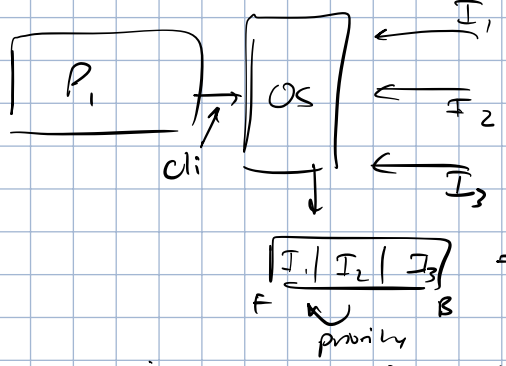
no value checks.

1 stack / thrd in kernel ⟹ 1 k-stack in total

```
handler () {
    kernel stack push  ⟹ check for malicious args
    ;
    ;  handle
    ;
    pop off stack
    return (errno, prio, thrd)
}
```

slow!

How to prevent interrupts:



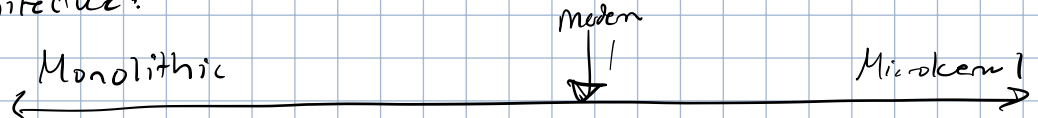
⇒ Buffr que.

B/c que size ⇒ can lose interrupts

Kernl has own virtual memory ← stack is in here

↳ Inacesible from process

Architecture:



Any thing in OS → kernl mode

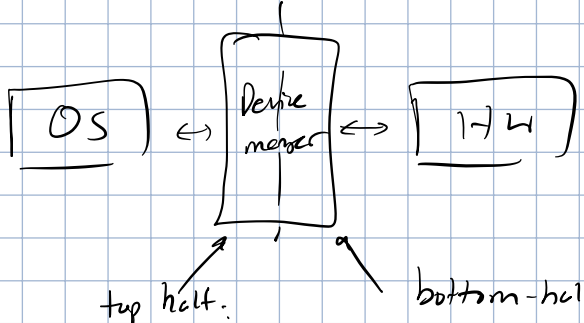
↳ Prob: failure! How to restore

Keep as little as possible in kernel mode.

↳ Error handling.

↳ Prob.: IPC in OS, not unified.

OS & HW



Q: how to transfer data

① Programmed I/O

Each byte needs CPU ⇒ CPU time ∝ size of a data.

Slow

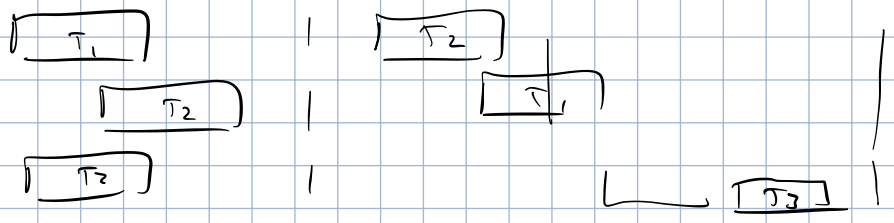
② Direct memory access (DMA):

Delegate to controller to transfer data ⇒ free up CPU

↳ Args: memory address, size

Put it into buffer

SYNCHRONIZATION & DEADLOCK



Possibility of conflict ⇒ race conditions

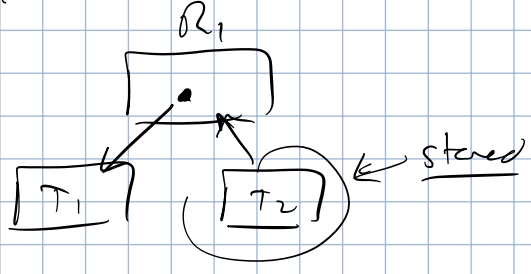
Atomic ops: load & store of bytes

Race Conditions

Sharing resources b/w threads → conflicts

↳ Order of threads exec matters

Starvation lockup:



Complication: compiler reordering instructions

↳ Assure this doesn't happen ⇒ sequential exec.

Not performant

Mutex

```
mutex.lock();
critical section => 1 thread runs at a time.
mutex.unlock();
```

will not be reordered

Tips:

① Everything shared in mutex → mutual excl.

② Thread that locks also unlocks.

OS-provided service

Issue: cannot help w/ scheduling constraints.

Semaphore

Non-negative int count w/ 2 atomic ops:

P():

1. Wait for semc. +ve } — comm
2. Decrement by 1

V():

1. Increment semapha by 1
2. Signal any waiting processes in P } — check

} Atomic ops. exec'ed by 1 thread at a time.

No thread will miss a wakeup call

Mutual exclusion: binary semphor (0 or 1)

```
sema.p()
    crit. sect.
sema.v()
```

Order matters → Hard to reason

Monitor

Monitor

Mutex | 0< condition variables.

Mutual excl. ⟶ Scheduling

Condition variable:

a) CV.wait(&mutex) ⇒ { ① Unlock your mutex ② Suspend thread ③ Lock again if woken up.

b) CV.signal() ⇒ Wakes up one of in ②

CV:

↑ NOT FIFO

c) CV.broadcast() ⇒ Wake up everyone in que.

Memoryless ⇒ no data associated w/ C.V.

Remember: re-enable threads in wait can't always run immediately.

Design path:

```
method_waits():
  mutex.lock();
  while(!testSharedState()){
→   CV.wait(&mutex);
  }
  .
  .
  .
  mutex.unlock();
```

```
method_signals():
  CV.signal();
```
] → lock held

Threads don't run immediately ⇒ state could have changed before lock acquired.

Implementation

Hoare
- Runs as soon as process wakes
- 'if' rather than 'while'

Mesa
- Always common
- Small bug: process can wake up w/ no signal

Since CV queue is not FIFO, don't depend on FIFO behaviour.

Semaphore useful w/ HW, which cannot access locks (could lock itself)

Communicating Seq. Proc.

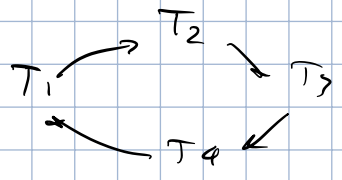
Style: no threads share data, use messages b/w threads instead.
↳ Don in channels.

Channels still need sync.

Deadlocks

Starvation ⇒ thread never gets to access data

Deadlock = starvation for all threads that need shared state
↳ Circular starvation.



Necessary, not sufficient conditions:

① Mutual exclusion: resource used by 1 thread at a time.

② Hold & wait: thread holds R_1, but waits on R_2

③ No pre-emption for resources: if T_1 holds R_1, R_1 can only be released by T_1

④ Circular wait

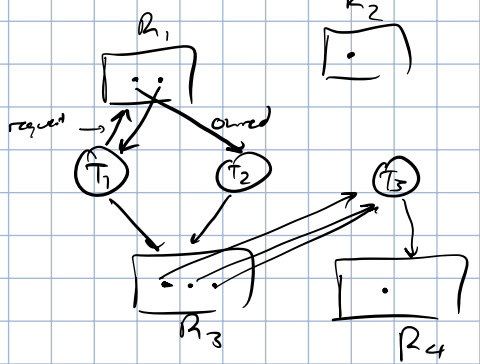
Deadlocks = f(timing, threads, resources, order)

Graphical view: resource allocation graph:

Threads: $T_1 \ldots T_n$

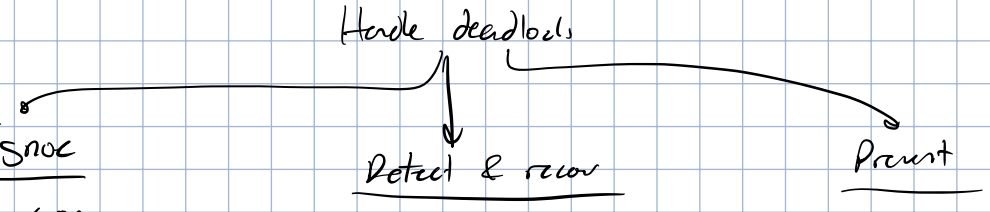
Resource types: $R_1 \ldots R_n$

Each type has instances:



Can help ⇒ circular path ⇒ deadlock.

Deadlocks are dynamic



Deadlock detection:

Assumptions:

m resources, n threads

Capacity: $[C_1 \ldots C_m]$

Free resources: $[F_1, \ldots C_m]$

Resource allows:

Threads ↓ Resources →

$$\begin{bmatrix} A_{1,1} & \cdots & A_{1,m} \\ \vdots & & \\ A_{n,1} & \cdots & A_{n,m} \end{bmatrix}$$

Resource requests:

$$\begin{bmatrix} R_{1,1} & \cdots & R_{1,m} \\ \vdots & & \vdots \\ R_{n,1} & \cdots & R_{n,n} \end{bmatrix}$$

Constraint:

$$\forall j, \quad \sum_{i=1}^{n} A_{i,j} + F_j = C_j$$

Also:

unfinished = $[T_1 \ldots T_n]$

```
do {
  done = true
  for each i in unfinished:
    if ([A_i,1 ... A_i,m] ≤ [F_1, ... F_m])
      remove i from unfinished
      [F_1, ... F_m] += [A_i,1 ... A_i,m]
      done = false

} while (done)
```

If unfinished is non-empty ⇒ deadlock (items in unfin are starving)

When to run:

a) Request made for resource.

b) Cannot get resource

c) Periodically

Deadlock recovery:

a) Terminate a thread ⇒ unblocked? → N → do it again

Issue:

1. Terminate a thread → inconsistent state.

2. Can run into same sit.

b) Proceed until can go further → exception handling.

c) Rollback to non-deadlock state.

Issue:

1. Requires for checkpoint

2. Same as a)

Deadlock Prevention:

Illegal, fix request order, grant resource @ beg. - - - -.

Don't request if deadlock goes to occur.

↳ Avoid unsafe state: ∀ scheduling order, threads may get blocked if max resources requested.

Ex: // 1 resource → 10 instances.

3 free instances:

Thread	Have	Max
A	3	9
B	2	4
C	2	7

Safe / unsafe?

Q: Order that allow each thread to get max resources?

① Allocate free to one w/ least diff. b/w "have" & "max"

Gr 2 → B

② Gr back max → free:

F: 5 resources

③ Repeat: B → C → A

Also: Banker's Algo

Same as deadlock algo except:

if ([Max_{i,1} ... Max_{i,m}] - [A_{i,1} ... A_{i,m}] ≤ [F_1, ... F_m]) { _ }

Before alloc

Slow & expensive.

MULTITHREADED KERNELS

① Context switch? User ↔ kernel?

When is switch?

a) System call
b) Processor exception } Sync
c) Interrupts → Async

We use handlers:

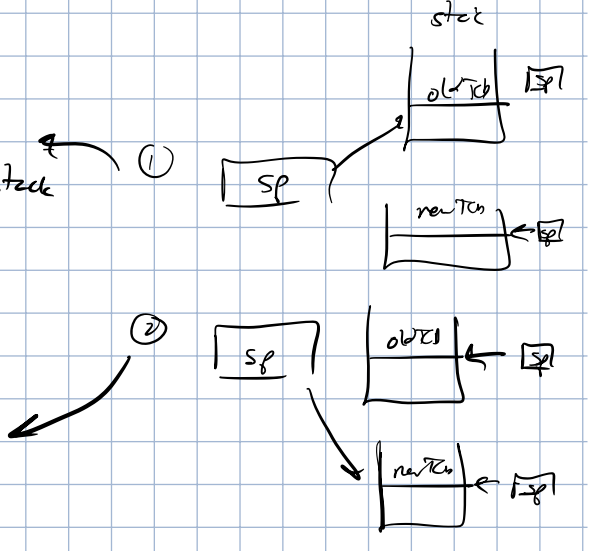
```
handler () {
    push regs to kernel stacks
    :
    : handle
    :
    exit:
      pop regs to user stack
      return
}
```

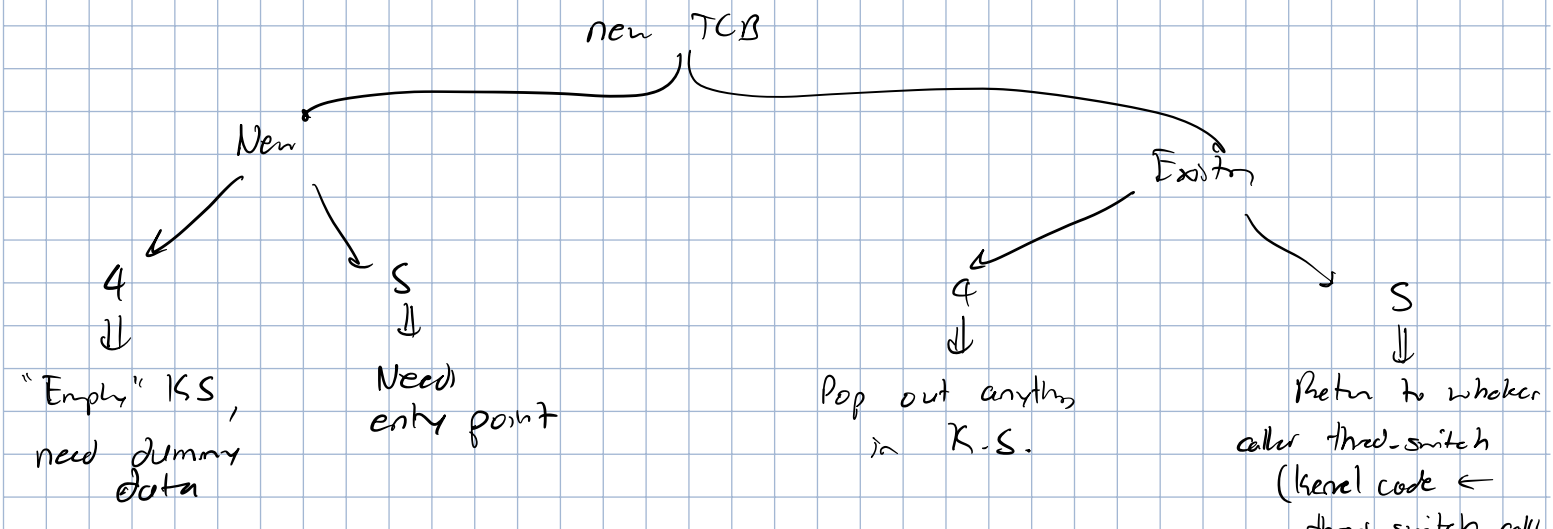
→ Switch mode / return.

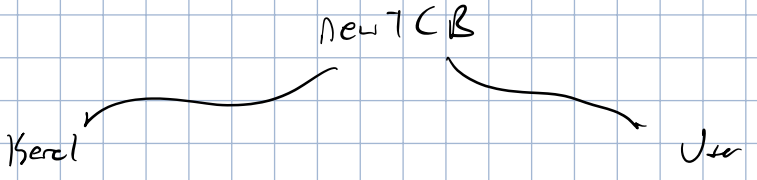
Thread switch func:

thread_switch (oldTCB, newTCB):

1. push all regs of oldTCB → kernel stack
2. oldTCB.sp = sp
3. sp = newTCB.sp
4. pop regs from kernel stack of newTCB
5. return







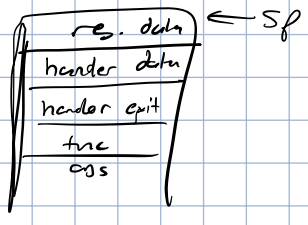
No mode switching necessary.

Switch mode from kernel → user
↳ Entry point can be special func. or handler from kernel

② Thread creation

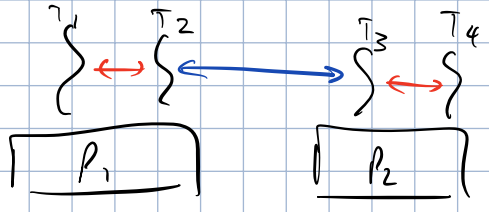
thread_create (func (entry point), args) :

```
TCB* tcb = new TCB();
tcb → sp = new stack() + stack_size →
push args
push func.
push dummy handle data
push handler exit point
push dummy data for regs.
set stack
add to ready queue.
```



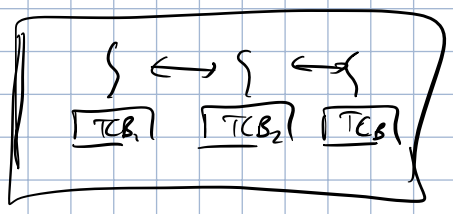
Will get out of switch & handle.

Kernel vs. User-Managed Threads



(T2 faster than T3) faster than CPU switch
↳ Reason: caches.

Kernel-managed threads:



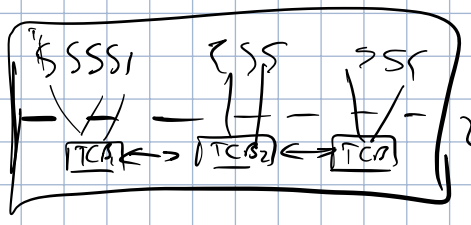
Kernel does all switching:

↳ 1:1

↑ Kernel threads

↑ threads switched by kernel

User-managed threads:



Kernel sees

Kernel doesn't know # of threads

∴ N:1

- Kernel switching has no effect on thread!
- Thread that runs can get blocked if TCB assigned not scheduled by kernel & thread waits I/O
- Cons:
  1. Not very parallel in multicore
  2. Blocking
     ↳ Scheduler activations: kernel messages → user → threads.

OS Classification:

# of addrs. spaces / # of thds	1	many
1	early OS	Unix
many	Embedded systems	Modern OS

Synchronization Implementation

① Mutex

Idea #1: disable all interrupts in mutex → prevent context switch

- Bad b/c user can do bad things w/out interrupts
  ↳ Too much code not under interrupt

Idea #2: restrict no interrupts on mutex:

```
Class Mutex:
  private:
    int value = FREE; // or BUSY
```

```
Queue waiting;  => true to access locked mutex
public:
    void lock();
    void unlock();
```

```
void lock():
    disable_interrupts;
    if (value is Busy):
        waiting.add(curTCB);
        runningTCB -> state = wait;
        schedulerTCB = schedule();
        thread_switch(runningTCB, schedulerTCB);
        <-- next_TCB here! returns
    else:
        value = Busy
    enable_interrupts;
```

```
void unlock():
    disable_interrupts;
    if waiting is not empty:
        nextTCB = waiting.pop();
        nextTCB = READY
        ready_list.add(nextTCB);
    else:
        value = Free
    enable_interrupts;
```

value remains BUSY, only nextTCB access.

Pros:
Reduced section in non-interrupt state.

Cons:
If blocked in lock() / unlock() => disable interrupts forever
↳ Soln: thread enable interrupt after thread switch
Still relies on disabling interrupts.

Idea #3: HLR atomic ops → spinlock → Mutex & scheduler.

```
test&set(&addr):
    result = Mem[addr]
    Mem[addr] = 1
    return result;
```

```
swap(&addr, reg):
    temp = Mem[addr]
    Mem[adr] = reg
    reg = temp
```

```
Compare&Swap(&addr, r1, r2):
    if r1 = mem[adr]:
        M[adr] = r2
        return
    else:
        fail
```

Spinlock:

```
class Spinlock:
    private:
        int value = 0;
    public:
        void lock() { while (test&set(value)); }
        void unlock { value = 0 };
```

Threads after 1st lock continually in while loop
↳ busy-waiting.

Pro: M.E Con: waste CPU

Scheduler & mutex:

```
class Mutex:
  private:
    int val = Free;
    Spinlock mutex_spl;
    Queue waiting;
  public:
    void lock();
    void unlock();
```

```
class Scheduler:
  private:
    Queue readyList;
    Spinlock scheduler_spl;
  public:
    void suspend (Spinlock * spl);
    void make_ready (TCB * tcb);
```

```
lock ():
  mutex_spl.lock();
  if value is Busy:
    waiting.add (runningTCB);
    scheduler -> suspend (& mutex_spl);
  else:
    value = Busy
    mutex_spl.unlock();
```

unlock spl ⇒ other proc. run.

```
unlock ();
  mutex_spl.lock();
  if ! waiting.empty:
    next = waiting.pop();
    scheduler.make_ready (next);
  else:
    value = Free
  mutex_spl.unlock();
```

```
suspend (& spinlock);
  disable_interrupts;
  sched_spl.lock();
  spinlock -> unlock();
  runningTCB -> state = WAIT;
  switch to next process
  sched_spl.unlock();
  enable_interrupts;
```

⇒ prob interrupt handler from accessing spinlock → infinite loop

Most mutexes free in comp ⇒ can use simple mutex impl. if no conflict, over. spinlock-based mutexes

② Semaphores

```
class Semaphore:
  private:
    int value
    Spinlock semaphore_spl;
    Scheduler scheduler;
    Queue waits;
  public:
    void P();
    void V();
```

```
void P():
  semaphore_spl.lock();
  if value == 0:
    waiting.add (TCB);
    scheduler -> suspend (& sema_spl);
  else:
    value --;
  semaphore_spl.unlock();
```

```
void V():
  semaphore_spl.lock();
  if waiting not empty:
    tcb = waiting.pop();
    sched -> make_ready (tcb);
  else:
    value ++;
  semaphore_spl.unlock();
```

Interrupt handler cannot use P()/V() b/c spinlock

③ Condition variables

broadcast: check to while

A: Basic implementation

```
class CV:
    Queue waiting;
    void wait(mutex*)
    void signal();
    void broadcast();

wait(mutex*):
    waiting.add(tcb);
    scheduler.suspend(tcb);
    mutex->lock();

signal():
    if (waiting not empty):
        thread = waiting.pop()
        scheduler.make_ready(thread);
```

B: Using semaphores?

1 / monitor?

```
wait(mutex*):
    mutex->unlock();
    semaphore.P();
    mutex->lock();

signal():
    semaphore.V();
```

This will suspend b/c only enter wait if state not correct → value in Semaphore == 0

Process will not wait if calls wait ← state in semaphore!

C: Semaphore / waiting thread:

ordered

```
wait(mutex*):
    sema = Semaphore();
    queue.add(semaphore)
    mutex.unlock();
    semaphore.P();
    mutex.lock();

signal():
    if queue not empty:
        Semaphore = queue.pop();
        semaphore.V();
```

Guarantees wait!!

UNIPROCESSOR SCHEDULING

Obj: workload of diff. tasks → optimal sequencing of tasks.

- Minor objectives:
  1. Minimize overhead.
  2. Work-conserve CPU (don't be idle)
  3. Pre-emption

Model:

Task: | CPU burst | I/O burst | CPU burst | I/O burst | ...

don't need CPU, longer → CPU burst

Metrics:

① Response time: $t_{finish} - t_{ready}$
↳ Not t_{start}

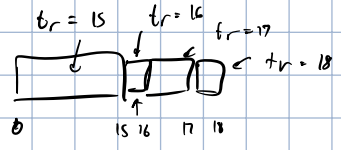
② Throughput: tasks/min
↳ Overhead
↳ Efficient resource

③ Fairness: CPU time shared

First Come First Serve (FCFS) Scheduling:

Pros: simple!

Cons: depends on submit time
↳ Short tasks stuck behind long tasks



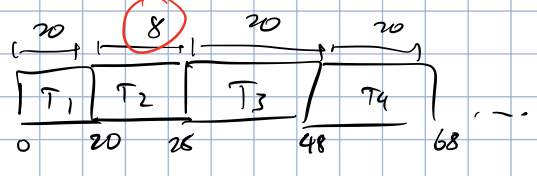
Best case: short tasks come first
Worst " : long task "

Round Robin (RR) Sched:

Time quanta → each task gets time quanta until expiration.

∴ N threads, q is quanta: max wait time = $(N-1)q$ (assuming overhead = 0)

Ex: $T_1 = 53$ $q = 20$
$T_2 = 8$
$T_3 = 68$
$T_4 = 24$



Wait time: $t_{start} - t_{prev\ finish}$

Response time: t_{finish}

Pros: fair

Cons:

1: Context switching overhead adds up $\Rightarrow \left(\frac{\Delta}{q+\Delta} < 1\%\right)$ c.s. time.

2: Depends on q: $q \uparrow \rightarrow$ response time $\uparrow$, $q \downarrow \rightarrow$ throughput $\downarrow$

Shortest Job First (SJF) Scheduling:

Heuristic: do task w/ least comp. first (CPU burst)

Variation: SRTF

↳ Preemptive!

Pro:

We can prove that SJF minimizes avg. response time.

Cons:

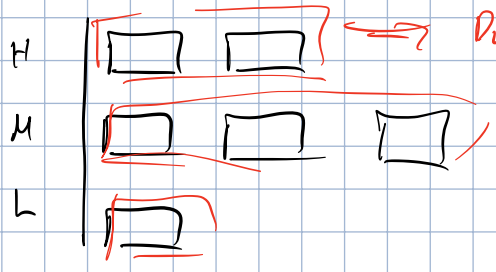
1. Long tasks can be starved
2. Context switching
3. Impractical → we don't know length of task.

↳ Prediction? Mistakes backfire.

Strict Priority Scheduling

Prioritize tasks w/ higher priority

↳ Strict: no higher priority waits for lower priority ⇒ pre-emptive

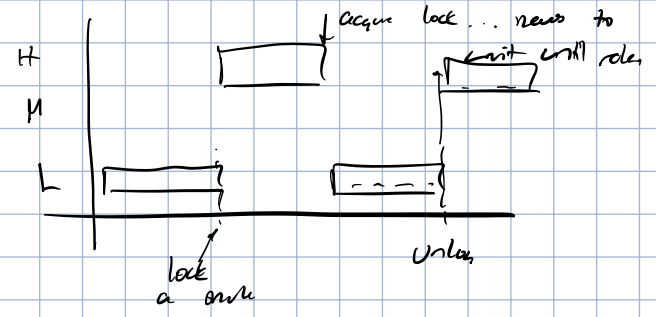


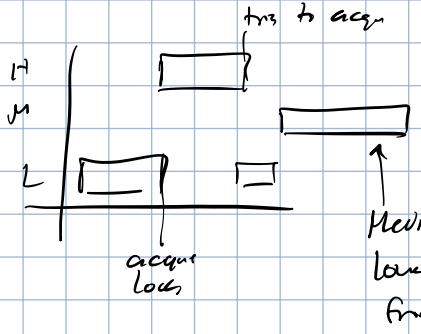
Do own scheduling policy within same priority

Problems:

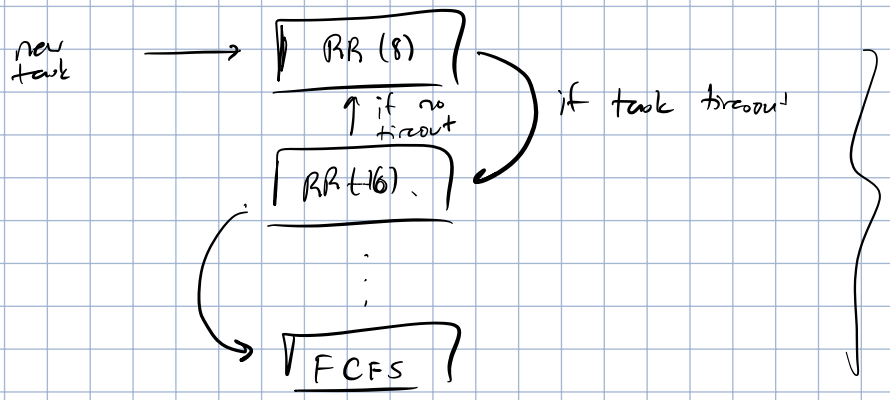
1. Starvation of lower pri task. ⇒ Soln:
   1. Time slicing b/w priority
   2. Decide long-running task pri
   } will affect response time.
2. Priority inversion: low priority delays high priority by holding shared resource.

Ex: //





Multilevel Feedback Queue



Pros:

1. Fair: short tasks run quickly, long-running tasks will run.

Lottery Scheduling

Steps:

① Give tasks some lottery tickets ⇒ # of tickets ∝ importance (priority, user)

② At each time slice → run lottery → run task w/ lottery #

Pros:

1. On avg: CPU time for task ∝ property of tickets task has
2. Prevent starvation: give every task 1 ticket
3. Works well in changing OS

Cons:

1. If too many short tasks, lose # of tickets → RR → response time
2. Randomness

Stride Scheduling

Obj: proportional time sharing w/out randomness

Algo:

① Stride per thread:

Stride: $\frac{W_i \Rightarrow \text{big } \#}{N_i \Rightarrow \#\text{ of lotto tickets (weight)}}$

② Maintain pass counter for each thread

③ On each time slice:

a: Find thread w/ least pass value (if tie → pick randomly)

b: Increment time w/ stride of selected thread

Ex: // $S_A = 100$ $S_B = 200$ $S_C = 40$

$P_A = 0$ $P_B = 0$ $P_C = 0$

t=0 (B) → $P_B = 200$

↓ 200

t=200 (A) → $P_A = 100$

↓ 100

t=300 (C)

↓ 40

t=340 (C)

↓ 40

t=380 (C)

↓ 40

t=420 (A)

↓ 100

t=520 (C)

} Proportional time sharing

Min-Max Fair (MMF) Scheduling

Obj: pick process w/ lowest acc. CPU time

1st impl:

① Look at specific quanta & run

② Switch to thread w/ low. acc. time.

} If too many tasks → RR → reopen time t

2nd impl: dynamic time quanta

① Set target latency (interval b/w task runs)

② Set time quanta:

$$q = \frac{\text{target latency}}{N \hookrightarrow \#\text{ of threads}}$$

③ Min. granularity:

pick max (q, min gran.)

} If N ↑ ⇒ too much context switching.

3rd: weighted max-min fair scheduling

① Assign w_i to thread i

② Time quota:

$$q_i = \frac{w_i \times \text{target latency}}{\sum w_i} \Rightarrow \text{Petra matters.}$$

③ Keep track of how much the acc. ⇒ switch at end of q_i

↳ Use virtual time: $\frac{\text{actual time}}{w_i}$

④ Switch according to lowest virtual time.

Why both strive? Dynamic quota → less context switching.

Linux is similar WRMMFS

Modelling Scheduling

① Det. modelling: pick workload & algo → compare perf.

Not practical ⇒ can't just "pick" workload IRL

② Queuing modelling: assume events happen in some distr.

Not practical for network events.

③ Simulation: run system

Basis of false comparison

ADDRESS TRANSLATION

Basics

Two resp:

① Protect memory: shared data / excl. data

② Allocate mem.

Units:

KiB ≠ KB, GiB ≠ GB, MiB ≠ MB

↑ power of 2 2^{10} ↑ power of 10 10^3

Size of system ⇒ dictate size of addressable contents

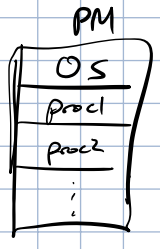
n bits ⇒ 2^n addr.

Protection

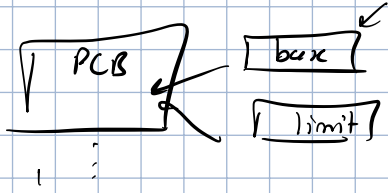
① Uniprogramming

1 proc @ time → entire phys. mem. for proc

② Multiprogramming:



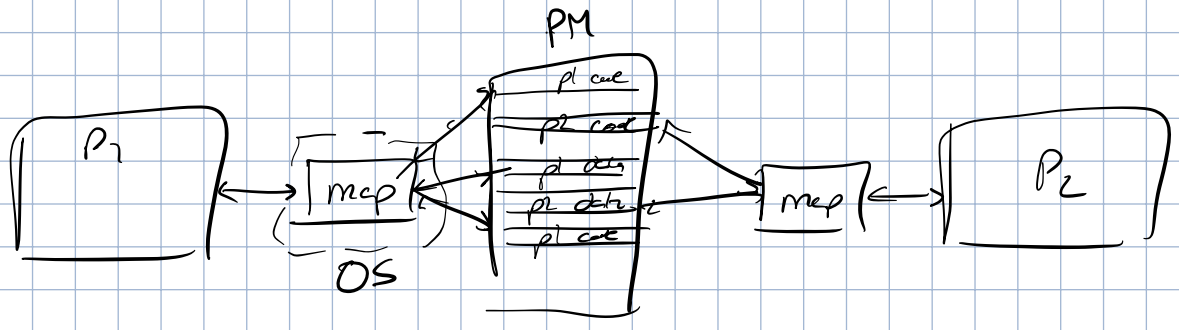
Directly use Phys. Mem



accessible mem ∈ [base, base + limit]

Issue: no sharing.

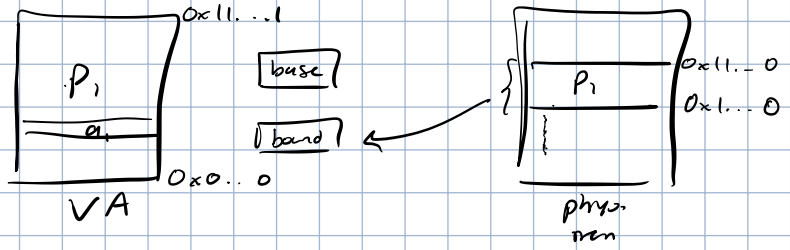
Translation



Pros:
- Illusion → easy dev.
- Protection
- Sharing

Cons:
- Overhead
- HW support

1st impl: base & bound.



Also:

① If a_1 ≥ bound ⇒ addr. too big ⇒ raise exc.

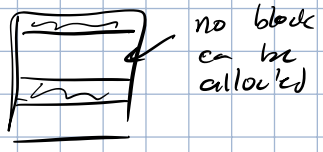
② PA = a_1 + base ⇒ access

Pros:
- Protection & isolation.
- Lower overhead

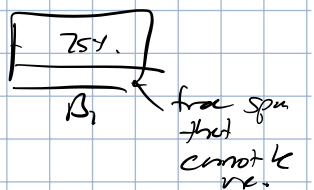
Cons:
- No sharing.
- Base & bound will shift (heap) → not good.

- Frag.

External:



Internal:

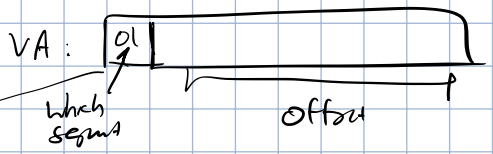


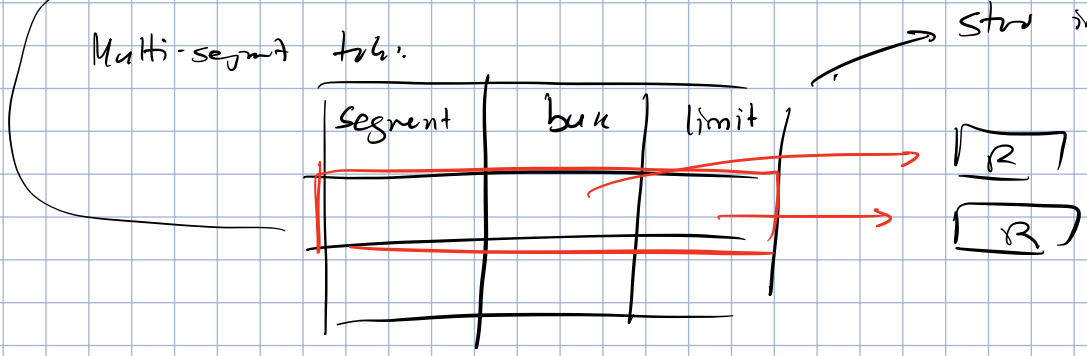
2nd impl: multi-segment addr. transl.

① Give each segment of proc. diff areas
heap, stack, env vars, ...

② Base & bound for each segment

③ Translation process:



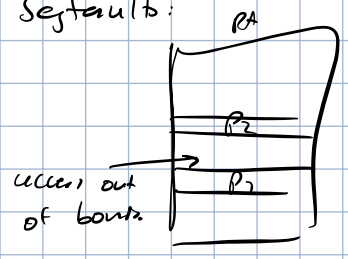


PA = base[segment] + offset if offset < limit

Pros:

① Reduce frag ← smaller segments

② Segfaults:



③ Easy for growing mem ⇒ do not have to copy mem.
↳ will cause segfault, but OS handle.

Cons:

① External frag.

② Swapping optimize.

Paging

Page: fixed size memory block (physical, virtual)

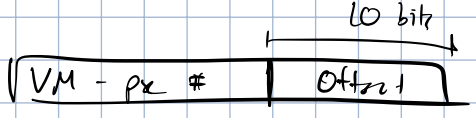
- Somewhat small (1 - 16 KiB)

Translation:

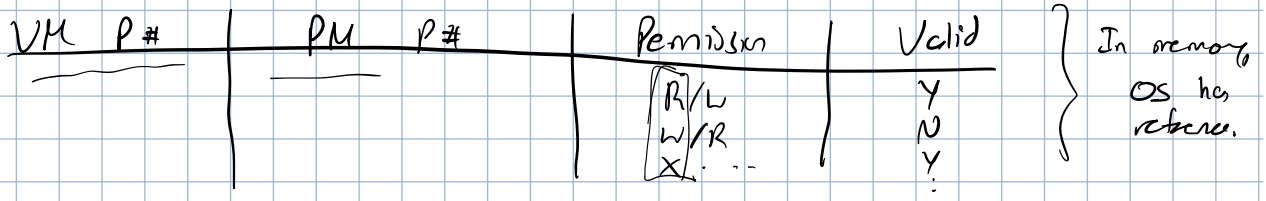
① Figure out size of page ⇒ # of bits to address a part of page

Ex: 1 KiB ⇒ 2^{10} bytes ⇒ 10 bits to address byte.

② Split virtual addr will have page offset & virtual page #



③ Table to match VM → PM page #:



Per process → virt. mem. changes / process

Heuristics:

1. Demand paging: only keep active mem. in table. Inactive → disk, invalid
2. Copy-on-write: on fork → copy PT & set read perms. If write → copy mem. block
3. Zero-fill-on-demand: set new data alloc to 0

Pros:

- Sharing is easy
- Page faults

Cons:

- Page table is huge: |page table| = # of VM pages.

Ex: 32 bit addr., 4 KiB page.

① # of bits to address byte.

$4 \times 2^{10} = 2^{12}$ ⇒ 12 bits in addr as offset

② # of bits for VM page:

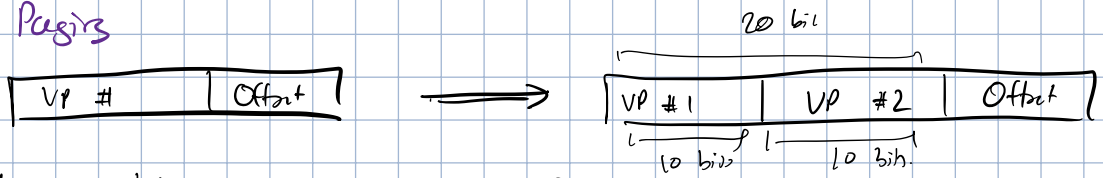
32 - 12 = 20 bits

③ # of VP:

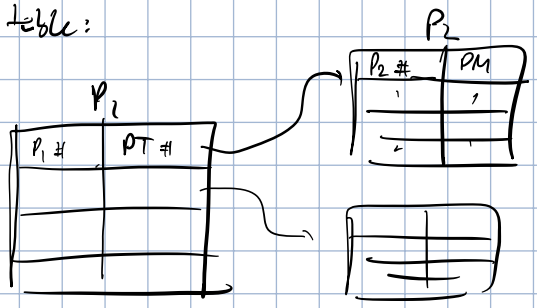
2^{20} page. If each PTE is 4 B

$\therefore 4B \times 2^{20} = 4\ MiB$

Multilevel Pagings



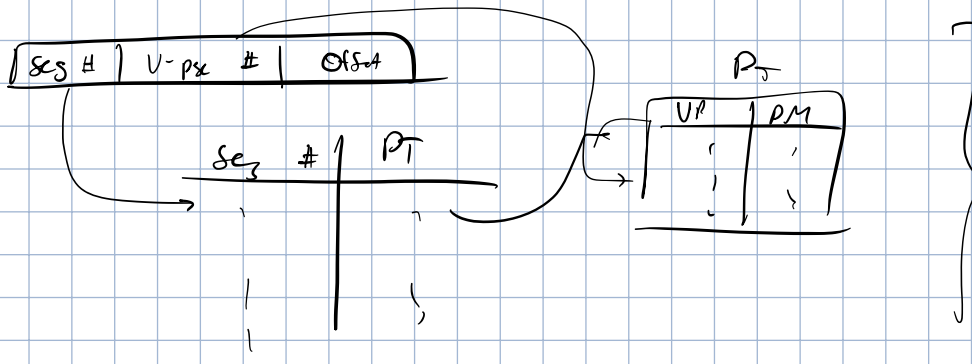
2 level page table:



This saves a lot of space:

If 1st level PTE not used → Associated P_2 is not there → 2^{10} pages saved.

Diff. flavor: segment pgtb



Share entire segment b/c easy to add chunks in PT

X86:
- 4 level pgtb → 2^9 entries.
- PTE: 64 bits

Shared libraries

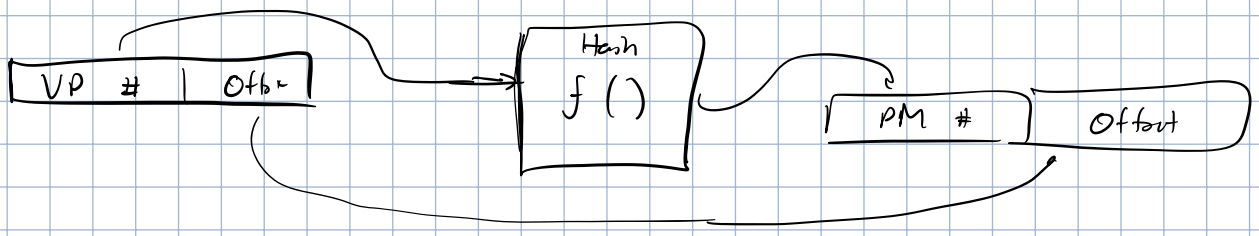
Processes should have own global & static variables to be used in shared libs
- On proc. switch → change value of vars.
- OS create global-offset table w/ all variables.

No absolute VA in shared library code ⇒ linked dynamically
- OS create procedure-linking table to go to procedures not in shared lib space.

Pros:
- Sharing
- Easy to alloc (page fault)
- Save space.

Cons:
- Overhead
- PT have to be contiguous in PM

Inverted Page Table



Sol: size of page table → no need to expand to address entire VM

Con:
- Poor cache locality (spatial)
- Collisions in PM

Hardware:

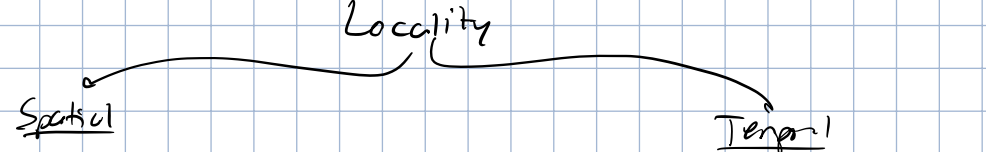
Most translation in HW → faster (MMU can detect page faults)

CACHING

Makes common case fast

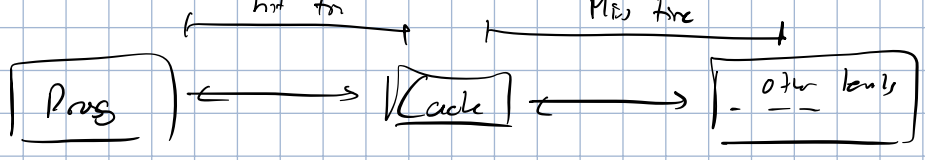
2 assumptions:
① Freq. case is frequent
② Infreq. case is not too expensive.

Locality



Can bring section of memory to cache to prevent prog to go to PM

Terminology:
- Block: consecutive bytes (unit of cache)
- Hit: access cache & find item
  - Hit time = time to hit − time to start access
- Miss: access cache & fail to find item



- Hit rate:

$$\frac{\#\text{ of hits}}{\#\text{ of hits} + \#\text{ of misses}}$$

- Avg. access time

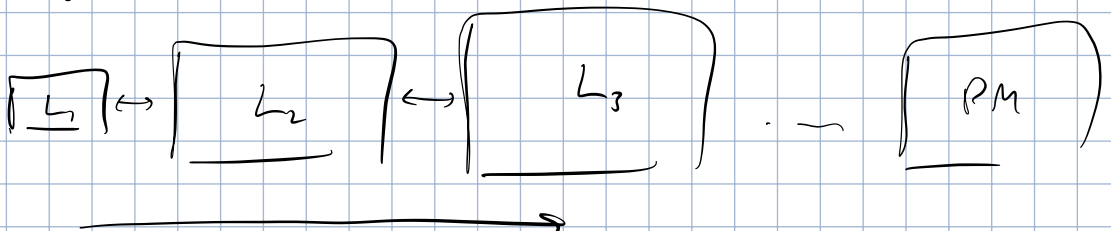
$$= \boxed{\text{hit time} + \text{miss rate} \times \text{miss time}} \quad \left(\text{hit rate} \times \text{hit time} + \text{miss rate} \times (\text{hit time} + \text{miss time})\right)$$

Tradeoff: low hit time & high miss rate.

Increase cache: miss rate ↓ ⟶ hit time ↑

Decrease cache: hit time ↓ ⟶ miss rate ↑

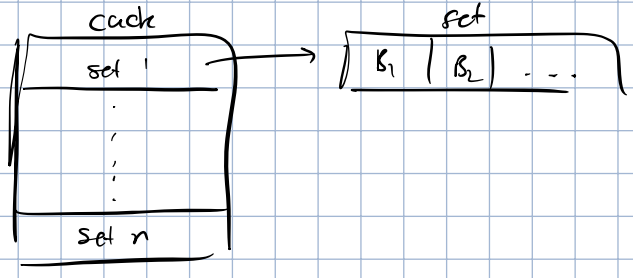
Soln: hierarchy of caches:



Avg access = hit time$_1$ + miss rate$_1$ × (Avg. time$_2$)

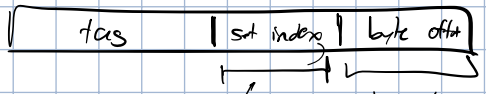
Avg. time$_2$ = hit time$_2$ + miss rate$_2$ × avg. access$_3$

Cache Mechanics



Associativity: # of blocks / set

To find block in cache:

Mem addr: 

log (# of sets) log (size of block)

$$\#\text{ of sets} = \frac{\text{size of cache}}{\text{size of blocks}} \div k$$

On access:

① Find set

② Find block by looking at tag

③ Find byte via offset

Ex:// 1 KiB, direct-mapped. 32 B blocks. Mem. addr?

32 B block ⇒ 5 bits for byte offset

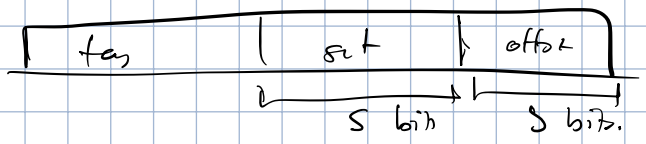
1 KiB cache, direct mapped.

$$\frac{2^{10}\ B}{\text{size of block}} = \frac{2^{10}\ B}{2^5\ B} = 2^5 \text{ blocks in str.}$$

Since D-M ⇒ 32 sets.

∴ Set index: 5 bits

Rest is tag:



Ex:// Same sit., 2-way associat:

Could store 2^5 blocks in str.

$$2^5 \text{ blocks} \times \frac{\text{set}}{2 \text{ blocks}} = 2^4 \text{ sets} \Rightarrow 4 \text{ bits.}$$

Prob: new memory diagrams has same set index → only use 1 set, cache is mostly empty!

↳ Soln: coloring → PA is 'colored' so that blocks of same color → same cache set

OS will make sure seen VA's from diff. colors.

Cache Miss & Eviction

Reasons

- Cold access
  - Never seen before
- Conflict
  - Cache is too small
- Coherence
  - Other procs update mem
    ↳ invalidate cache

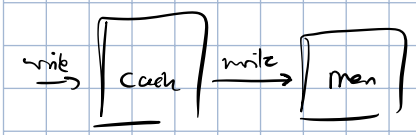
Eviction:

- Direct mapped: kick out block
- n-way associative: LRU, random choose

Cache Writes

Write methods

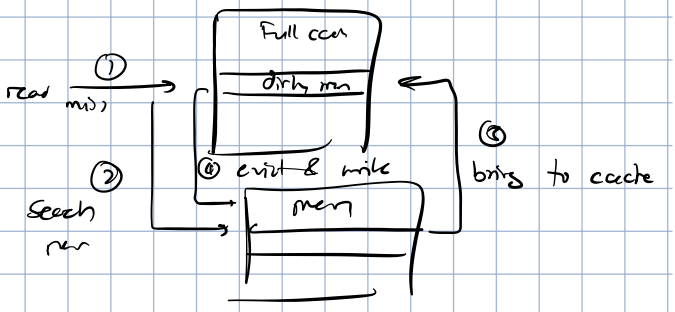
Write-through



- Pros: read misses will not create writes.
- Cons: processor is slow → write buffer.

Write-back

- On write → modify cache to signal it's dirty
- On read miss & replace → write to memory



- Pros: no write hold up
- Cons: overhead on read miss

Address Translation

Cache VA ↔ PA translations ⇒ TLB (translation lookaside buffer)

↳ If miss, check PTE

Q: What if address perm change

Perm ↑ ⇒ nothing.

Perm ↓ ⇒ flush out translation.

Q: How are writes handled?

Use a dirty bit in TLB.

If bit == 0 ⇒ write through on bit (set dirty)
If 1 ⇒ don't have to write through

→ OS can emulate dirty bit

If read → update if necessary

Challenges:

① Homonym

B/w diff processes: 1 VA → many diff. PA

Soln: add proc. ID on TLB entries to print many PA usages.

② TLB Shoot down

TLB / processor ⇒ 1 processor updates TLB → refresh in all oth proc. TLB

Need to send inter-processor signal to remove entry if update.

③ TLB miss

Expensive ← Page table walks!

Q: What do we do if we access invalid page / page is not in PT?

A: Hardware for PTW

Raise interrupt → OS will trap to page fault handler

B: Software for PTW

Page fault raised → Kernel will do PTW → page fault handler

Challenge: out of order execution

Ex: 1. mul r1, r2, r3
2. bne r1, r4, loop
3. ldr r5, (r6)

Issue: compiler may order 3 b/f 2
if 3 faults → should 2 be done again? Out-of-order exception.

Soln: use precise exception

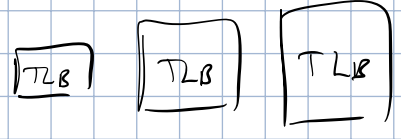
① Keep track of retired instr. (state is in HW)

② Only handle exception if instr. that raises is retired
↳ all prev. instructions should complete.

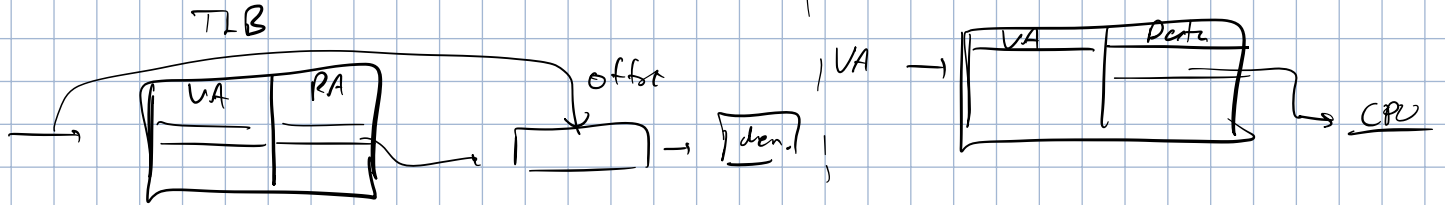
↓
In architectural state (regs., mem.).

Speedup of TLB?

① Multiple levels of TLB



② Virtually-addressed cache:



Why speedup? No translation!

Challenges:

1. Synonyms: diff VA → same PA

   Cache stores multiple KVP w/ same VA! ⇒ Aliasing

   Soln: tag entry in cache w/ PA & enforce same VA

   ↳ How to translate VA → PA via TLB → check if correct data in cache

   This problem happens if you depend on VA in any DS

③ Bring PTE to cache

   Bumb going to mem.

④ Set associativity:

Avg mem access time = hit time$_{TLB}$ + (miss rate$_{TLB}$ × miss-time$_{TLB}$)

(hit time: Direct mapped ↓; Fully associative ↑; miss rate: ↑; ↓)

Cost of miss is really high → fully associate.

⑤ Super pages

TLB will now address bigger pages → don't need + have more entries!

↳ Consq: more offset bits needed in VA

Need a bit to indicate whether PTE is superpage

# of PTE ↓

Linux

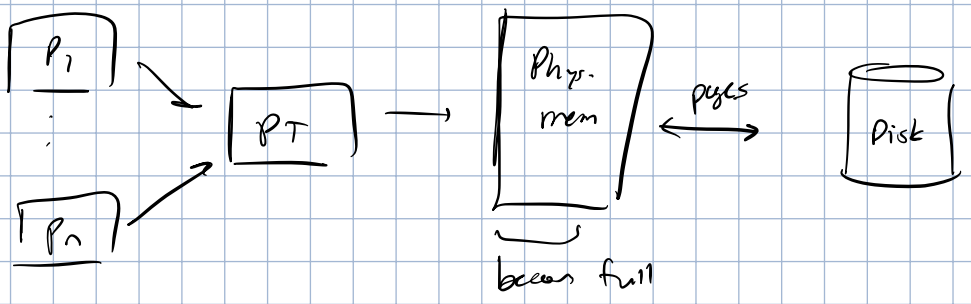
Issue: TLB filled w/ user proc. translations on kernel switch

Soln (subpar): kernel all in same addr, no need to flush TLB

↳ Security concerns

DEMAND PAGING

Intro



Physical mem. acts like cache!

- Block size: 1 page
- Associativity: fully-associative
- Write-back cache upon eviction

Switching is v. expensive, switching pages is a background

Q: How do we know if page is switched out?

Valid bit in PTE (1 if in PM, 0 if in disk) → causes trap on all entries.

This valid bit is updated on eviction

If we try to access → raise page fault → OS will bring back

Backing store / swap file:

Disk will mirror all virt. pages

Q: How do we bring back pages into P.M.?

OS will keep track of non-resident V.P.

Q: What causes misses?

Cache miss

- Compulsory miss (never seen before) — unavoidable, optimize bootup
- Capacity miss — Not enough space in P.M. — Soln: increase memory
- Conflict miss — N/A
- Policy miss — Page kicked out prematurely — Soln: use better policies.

Page Replacement Policies

① Random eviction
- + Simple
- − Not efficient

→ Used in TLB

② FIFO: kick out oldest
- + Fair
- − Oldest page could be well used.

③ Minimum: kick out pg not least in future.
- + Optimal
- − No idea about future.

④ LRU: kick out page use least in past
- + Good approx of min., but not necessarily best.
- − Overhead

Belady's Anomaly:

Thought: just increase amount of memory → drops miss rate?

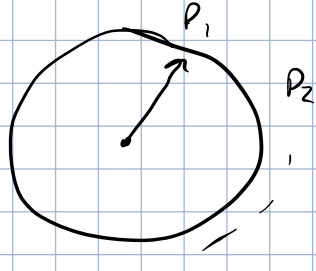
Ans: NO! In certain policies → increase size, increase in page fault (FIFO)

LRU: Clock Algorithm

LRU w/ L.L. has overhead.

Clock also:

① Arrange phys. pgs in circle.



② Whenever we evict, we invent hand & examine "access bit" of page: (TLB/PTE)

If access = 0 (it is not access) → kick out

If access = 1 ⇒ set to 0 and move to next page.

③ If all access bits = 0 ⇒ FIFO policy

If hand is moving fast ← lot of page faults.

LRU: N^th Chance Algorithm

① Set variable/page to record # of times it could be evicted.

② If hand is at page & access bit is 0 → increment variable.

③ If variable = threshold → evict

④ If access bit is 1, reset to 0

Threshold: # of chances that we will give before eviction

- Too high: it will take too much time.
- Too low: not good approx.

Pro: better approx. of which pages are least used.

If page is dirty: write back & give 1 more chance.

Q: when to run?

① On page fault (add. overhead on page fault)

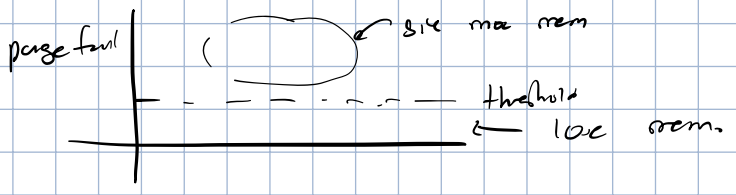
② Run async

Memory Allocation

Fixed Allocations strategies:

- Equal allocation
- Proportional alloc.
  - Alloc. mem. based on size of process
- Priority alloc:
  - Proportional w/ priorities.
  - On replacement: take mem. from lower pri.

Adaptive alloc.: give mem according to page fault



Thrashing: process busy swapping mem. from each other

Soln:

① Detect

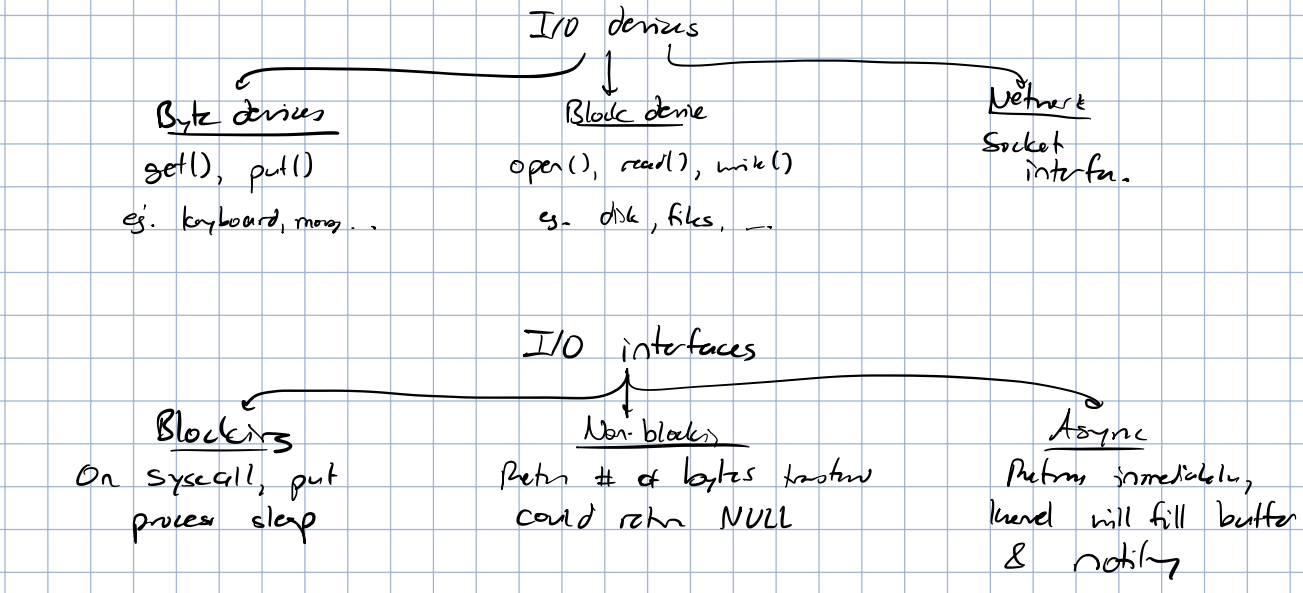
② Swap out entire processes.

Working set: set of page accessed frq. in past few ITs

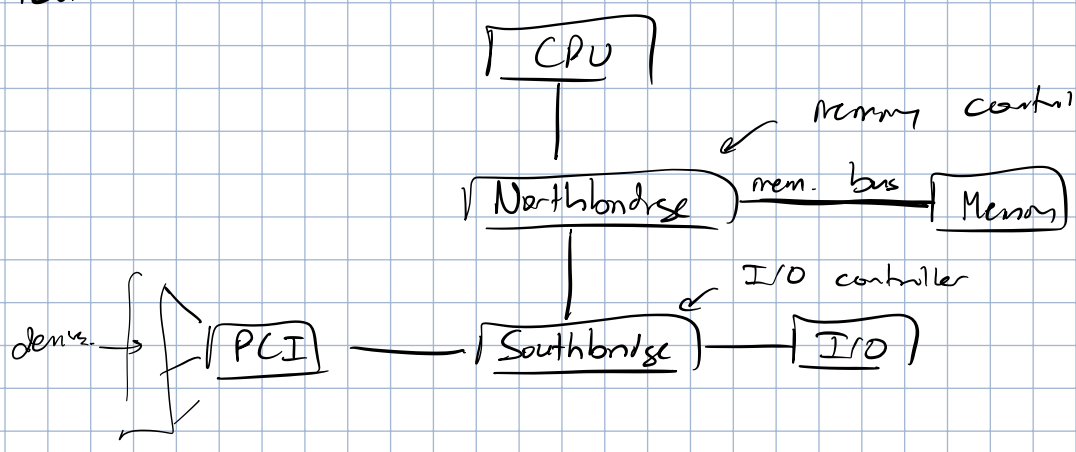
↳ Big enough set into memory → hit rate ↑

↳ If too many ⇒ thrashing likely so swap out procs.

I/O



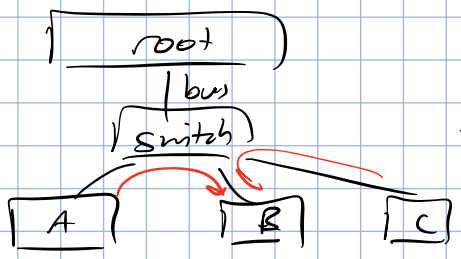
Architecture:



PCI & Devices

Originally a parallel bus → limited by speed of slowest device.

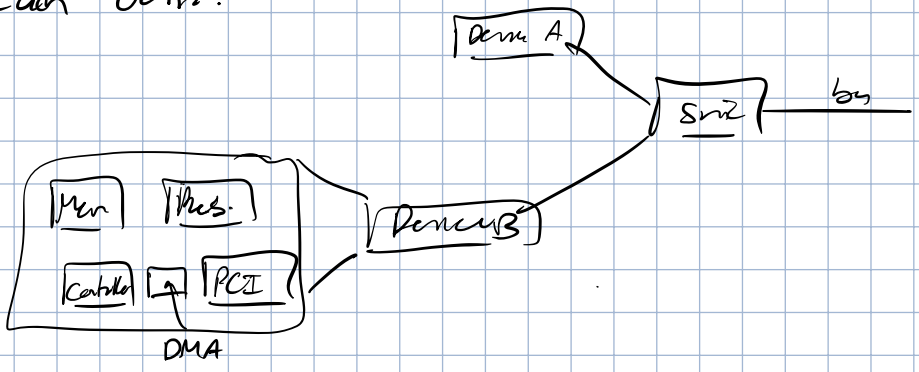
PCIe:



→ select-based, faster!

Good backward compatibility

Each device:



On BIOS, devices will tell OS & auto-reg.

OS ↔ I/O access

Port-mapped

I/O device has a sep. addr. space.

CPU uses special instr.

Memory mapped

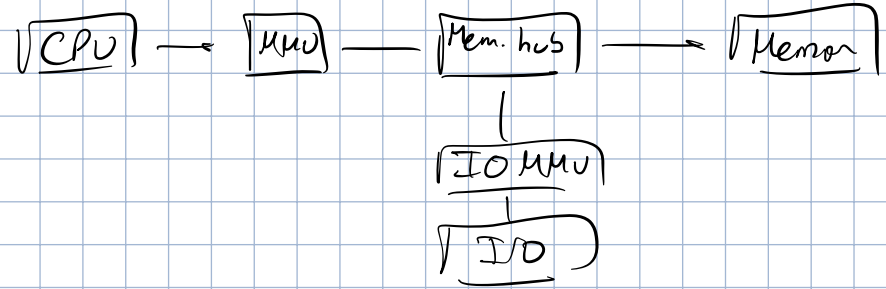
Part of physical mem → I/O device

OS can use memory ops.

need protection!

We need I/O to interact memory

↳ For security → address translations for I/O



2 problems:

① I/O can bypass IOMMU by caching

② IOMMU not used for p2p device comms.

Soln: access control series forces all device packets to go through IOMMU

I/O Performance

Peak bandwidth: max rate of data transfer
↳ Depends on HW

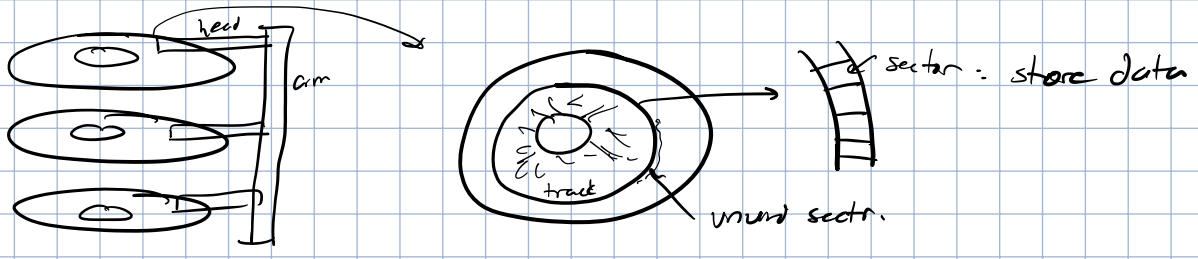
Effective bandwidth: actual rate of data transfer.

↳ Latency ↑ → size of data ↑

$$\text{Latency}(n) = \text{overhead} + \frac{n}{\text{peak bandwidth}}$$

$$\text{Effective bandwidth}(n) = \frac{n}{\text{latency}(n)}$$

Disks



Outer half of disk is used

Process of reading data:

① Seek track → move head (seek time)

② Rotate disk to before sector (rotational latency)

③ Transfer blocks b/w sector & head (transfer time)

Eg:// Seek time is 5ms. Disk rotates at 7200 RPM. Transfer rate of 4 MiB/s w/ sector size of 1 KiB.

a) How long to read sector from random sector?

① Seek time → 5ms

② Rotational latency:

$$60000 \frac{\text{ms}}{\text{minute}} \times \frac{\text{minute}}{7200 \text{ rotations}} = 8 \text{ ms/rotation}$$

③ Transfer time:

$$2^{10} \text{ B} \times \frac{\text{s}}{2^{22} \text{ B}} = 0.24 \text{ ms/B}$$

Add: 5ms + 4ms + 0.2ms = - - -

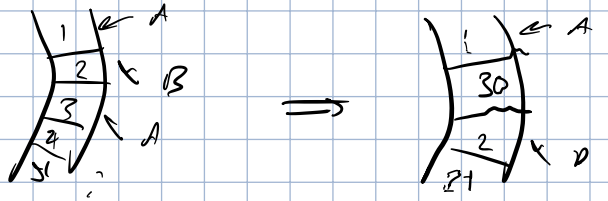
Sequential reads are fast → cut out seek time & no rev to rotate

Error correction:

① Sector sparing: send corrupt data to spare sectors.

② Slip sparing: remap sector → sequential reads

③ Track skewing: offset sector # to allow for seq. reads



Disk Scheduling

① FCFS

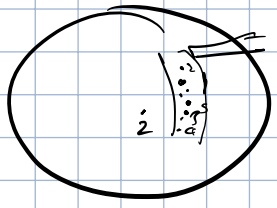
Pros: Simple

Cons: Poor performance → keep switching tracks, create

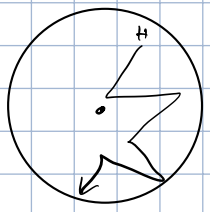
② SSTF: pick request w/ shortest seek time (closest to head)

Pros: save on time

Cons: starvation of tasks far away from head



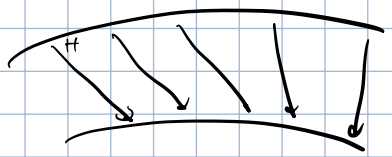
③ SCAN: arm moves in 1 dir → spin the disk to pick closest request, move up & down in disk



Pros: low seek time & no starvation

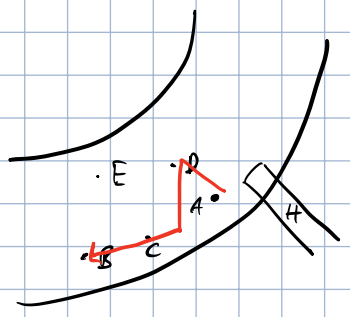
Cons: unfair to inner/outer requests

④ CSCAN: arm moves but switches to top



Much fairer than SCAN

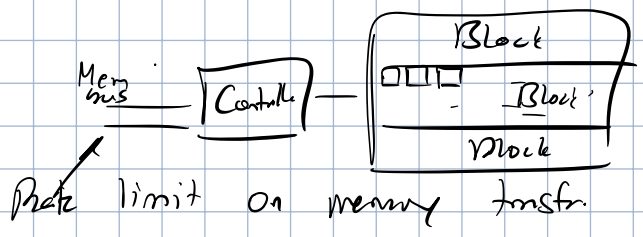
⑤ R-CSCAN: take rotational delay



→ Move up & down to read requests if < rotational delay to get to next rec. in trac

Flash Memory

Better hard drive ← more efficient, no moving parts



Latency overall = queue time + controller + transfer bytes.

Con: writing

- Can only write to empty pages. Clearing empty pages → clear whole block
- Remap memory using flash translation layer (FTL)

Flash has garbage collector → keeps blocks together if stored @ same time.

FILE SYSTEM

Intro

Abstraction for persistent, named data

persistent: always stored unless deleted

named: address data by human names

Goals:

① Naming

② Data-storage management

③ Protection

④ Reliability / durable

File

Data — Metadata

- Bunch of blocks

Characteristics:

① Mostly small files, grows over time

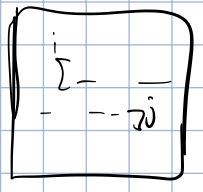
② Most space taken by big files

File access

Pattern:

① Sequential access: read bytes in order

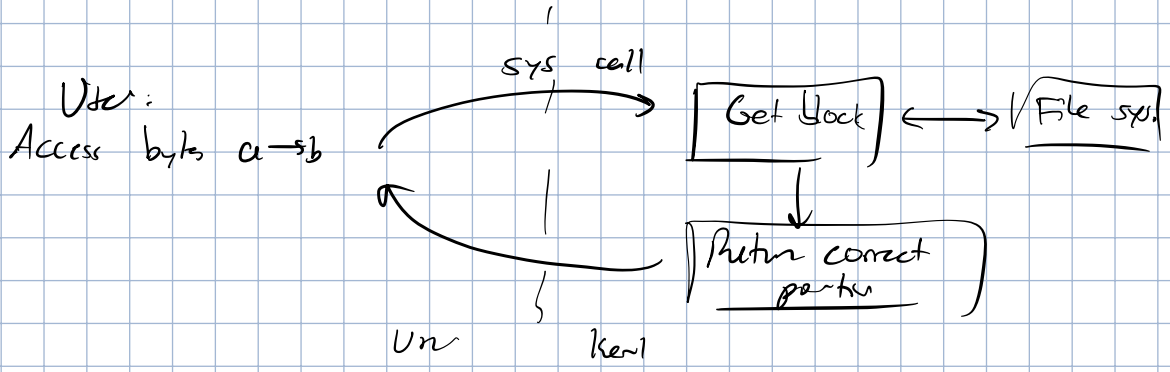
② Random access: can read anywhere in file



→ Should be fast

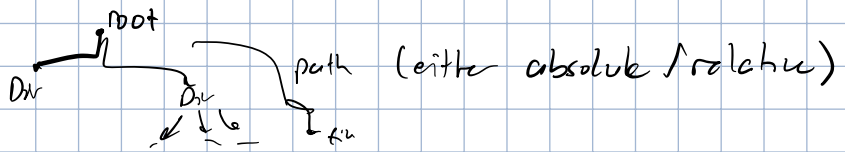
③ Content-based access

Eg: grep, filtering → need indexes



Directory

Defn: list of human-read names ↔ files

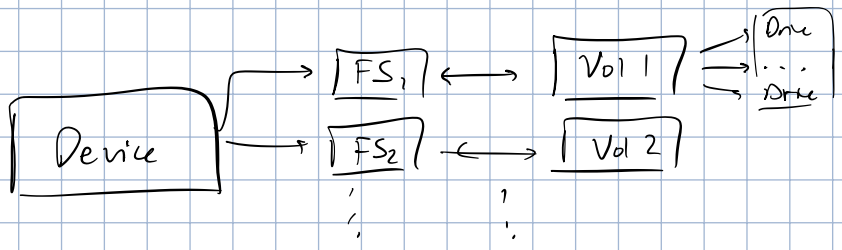


Can create links, which are alt. paths to file. File is not deleted if hard link exists.

↳ Alternatively, use soft links which can point to nothing

Stored like a file

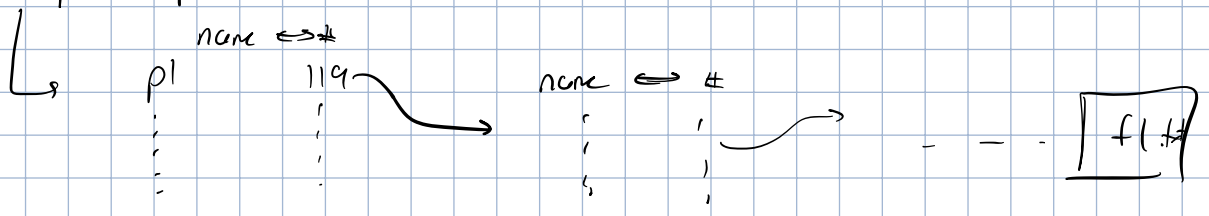
Volume



Device volume file systems will handle files by mounting to device file system

Search

/p1/p2/.../f1.txt



Implementation:

① Linked list

Name	dir	dir	file
File #	—	—	—
Next	↳	↳	

+: Simple
−: Slow if lots of files

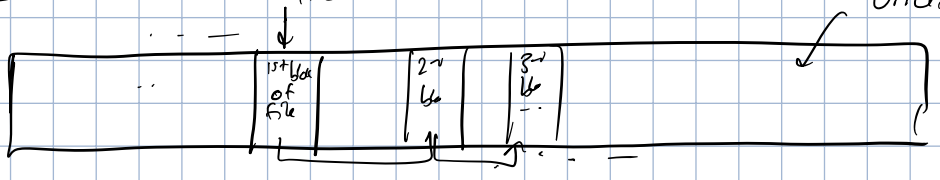
② B+ tree

Fast, b/c logn search / level

Block Access

① Find file # ⟶ ② Get block to access (FAT table)

FAT table



Q: Grow a file?

Easy, just link empty block to end block

Q: Where is it stored?

In disk → bring to memory & cache on boot

Q: Format disk?

Just clear FAT table

Pros:

+ Extremely simple

Cons:

- Random access is slow
- Poor spatial locality: blocks can be all over FAT for 1 file
- No hard link support: no metadata stored
- Limited volume & file sizes
- Security issues