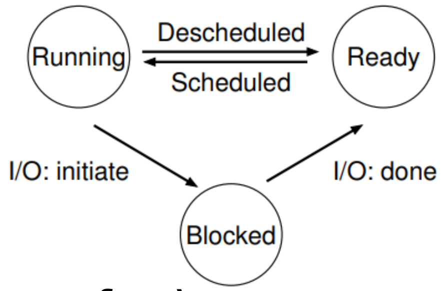
chapter 4 - process

prozess: ein gestartetes programm
virtualizing: jeder prozess glaubt seine eigene cpu zu haben (**time sharing**)
context switch: wechsel von einem prozess auf einen anderen, dabei wichtig: register sichern, reg für neuen prozess wiederherstellen
mechanisim: wie soll etwas(zb **context switch**) geschehen
policy: was genau soll geschehen?(zb welcher prozess kommt als nächstes dran?)
process api: create, destroy, wait, suspend(a process)
process states: running, ready, blocked(zb bei I/O request)
zombie process: fertiger prozess, der zb auf statusabfrage durch parent wartet



chapter 5 process API(Application Programming Interface)

fork(): "klont" den aufrufenden prozess. hat die selben(kopierten) variablen,
| register wie der elterprozess. return child id bzw. 0 (im child)
wait(): wartet auf statusänderung(zb exit) von child
exec(): transformiert prozess indem der programm code überschrieben wird

chapter 6 - mechanism: limited execution

create a process:

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

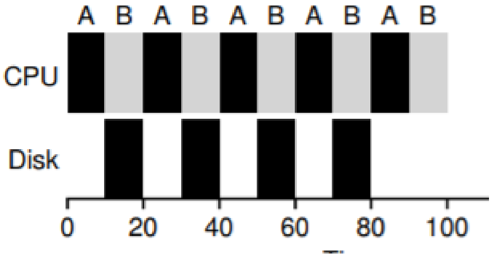
user mode: restrited, zb no I/O
kernel mode: mode the os runs in, alows privileged operations, full access to all recources
system call: (user) programm executes **trap** instruction to raise privilege lvl to **kernel mode** (if allowed) and perform instruction(zb I/O) when finished OS performs **return-from-trap instructuion**
-> switch back to calling user programm and user mode
boot: OS bootet im kernel mode und richtet **trap tabel** ein
trap tabel: enthält info was bei **exceptional events** passiert

switching processes: coroparative vs non coroparative approach
waits for syscall(zb I/O) || regelmäßiger **timer interrupt** by OS

limited direct execution: run pro on cpu, but limit what it can do wihtout OS

chapter - 07 scheduling

turnaround time: T-completion - T-arrival
response time: T.firstrun - T.arrival
(FIFO)/(FCFS)FirstInFirstOut/FirstComeFirstSever: bad turnaround time
Convoy effect: short jobs queued behind a "heavy weight" job
(SJF)ShortJobFirst: hilft nur bei zeitgleichem arrival mehrerer prozesse
(STCF)SHortestTimeToCompletionFirst/: umgeht convoy eff, da lange jobs zugunsten
Preemptive Shortest Job First(PSJF) : kürzerer pausiert werden
(RR)Round-Robin: better response time, worse turnaround time
time slice: multiple of timer interrupt, to short->time wasted by context switch
fair policy: better in response, worse in turnaround time
overlaping:



chapter 08 - Scheduling: Multi Level Feddback Queue(MLFQ)

THE CRUX: HOW TO SCHEDULE WITHOUT PERFECT(a priori)KNOWLEDGE(of job lenght)?
->queues with different priority levels

Basic Rules:

- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn’t).
- **Rule 2**: If Priority(A) = Priority(B), A & B run in RR.

How to change Priority ?

- **Rule 3**: When a job enters the system, it is placed at the highest priority (the topmost queue).
- ~~Rule 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).~~
- ~~Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level.~~

Problems:

Starvation: Jobs at bottom queue never complete if to many new jobs come in
Game the scheduler: running 99% of a time slice on purpuose -> never drop in lower queue -> monopolize the cpu
What if process changes behavior? -> cant come up the queue

Priority Boost:

- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.
  -> solves starvation and behavior change
  What time period for S?**(voo-doo constant)**

Gaming Tolerance(new rule 4)

- **Rule 4:** Once a job uses up its **time allotment** at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Tuning MLFQ and Other Issues:

how many queues?, time slice lenght per queue? priority boost frequency?
-> no easy awnser/depends on running processes

chapter 09 - Scheduling: Proportional share

CRUX: HOW TO SHARE THE CPU PROPORTIONALLY
How can we design a scheduler to share the CPU in a proportional manner?

tickets: represent the share of a recource that a process should receive
lottery: by knowing how much tickets a process has and how much there are in total the scheduler can pick(random) a winning ticket
advantage: random can be very quick
ticket currency: allow user(process?) to **transfer** tickets between own jobs
ticket transfer: allows one process to transfer its tickets to another one
usage: client/server setting -> c passes tickets to s to speed up its request
ticket inflation: raise/lower own tickets(only usefull if pros"trust"each other
fairness metric: T.job1.completes - T.job2.completes ([0.5, 1.0])
stride scheduling: each pro has a stride(more tickets->smaller stride)
+ pass counter, pro runs->increase pass counter by stride, pro with lowest pass counter runs. Completely fair after full cycle, but global state
-> what to do when new process comes in queue
(CFS)CompletelyFairScheduler: efficient fair-share (linux) scheduler
virtual runtime: pros accumulate vr by simply running on the cpu
scheduling decision -> run pro with lowest **vruntime**
problem: pro with long I/O has much lower **vruntime**->monopolzes cpu for some time
solution: CFS sets vr to lowest cr found in the **Tree**

CFS prameters

sched_latency: length of a time slice (e.g. 48 ms)
min_granularity: min length of a time slice (e.g 6ms)
Niceness: can be set by user for each process
range: (+19) - (-20), default 0, positive values -> less cpu time
Red-Black-Trees: balance tree, where running or runnable processes are kept in, ordered by vruntime. serching,insert,delete are O(log n)

chapter 13 The Abstraction: Adress Space

multiprogramming: multiple processes are ready to run at the "same" time
Adress space: Speicherbereich der einem Prozess alleine zur verfügung steht
stack: function call chain, local variables, parameters, return values
heap: dynamically-allocated, user-managed memory (malloc(), new)
transparency: realising **VM** without the pro "noticing"
VM virtual memory: 3 goals: transparency, efficency, protection
microkernels: **isolate** pieces of the OS from other OS pieces -> more reliability (one part can fail without affecting the other one)

chapter 14 Memory API

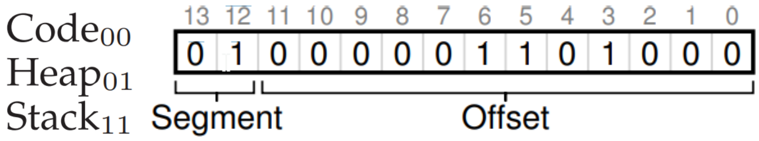
malloc(): returned pointer auf heap speicher stück der angefragten größe
free(void *ptr): gibt das heap speicherstück frei, auf das der pointer zeigt
brk/sbrk: used by the memory allocation library. used to change location(adress) of the end of the heap
calloc(): wie malloc(), aber "nullt" ihn
realloc(): vergrößert von malloc() allokierten speicher
dangling pointer: pointer auf einen wert, wobei man auf den pointer keinen zugriff mehr hat der pointer "schwebt" unerreichbar für uns herum

chapter 15 - adress translation

(hardware based) adress translation: umwandlung von virtueller in physische scpeicheradresse
dynamic relocation: adress space wird verschoben
base and bounds: hardware register die die virtuelle adresse berechnen/kontrollieren
static relocation: loader rewrites all adresses (no protection, kann nur schwer verschoben werden)
memory management unit (MMU): part of the processor that helps with adress translation

chapter 16 Segmentation

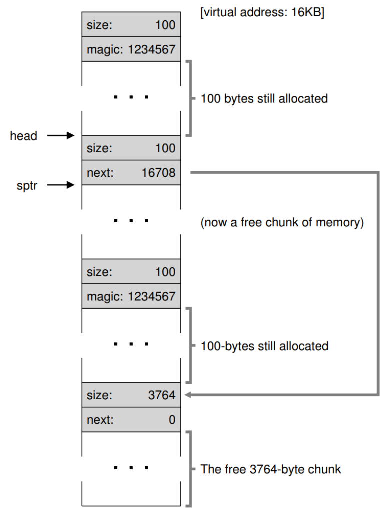
problem: large adress space(e.g. 4gb) but program only uses a few mb(large free space between heap/stack
segmentation: storing code/heap and stack on different locations -> no wasted space between heap/stack
-> 3 base and bounds registers, one pair for each segment
explicit approach:
segmentation bit: zusätzliche bits vor der virtuellen adresse um das segment(code/heap/stack) anzugeben
Mask: acces segment or offset bits via &



implicit approach: no segment bits, determines the segment by noticing how the address was formed
program counter -> code/ based on stack pointer -> stack/ other -> heap
sharing/code sharing: share code between multiple pros
protection bits: zusätzliche bits um protection zu regeln (lesen/schreiben/ausführen)
coarse-grained vs fine-grained: fine: mehr segmente, braucht zusätzlich segment table
external fragmentation: schweizer käse
solution: compact physical mem by rearanging existing fragments
nachteil: vergrößerung eines segments wird schwieriger

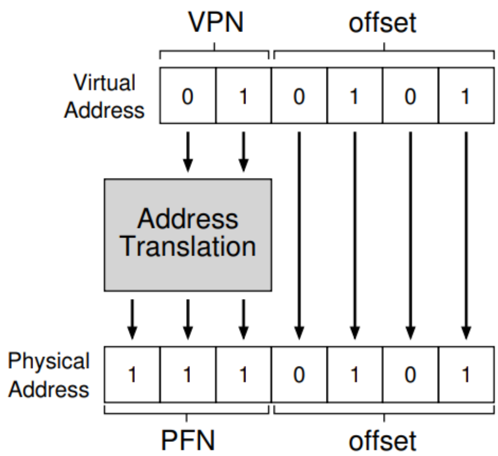
chapter 17 Free-Space Management

problem: segmentation causes (external) fragmentation: free space is chopped into little pieces
free list: manage free space on heap (struct node { int size; struct node *next;})
internal fragmentation: allocator hands out bigger chunks than requested
compaction: fight fragmentation by relocating chunks. dont work in heap because not all pointers are known
splitting: split free chunk to fit requested size
coalescing: when returning space (free) check for nearby free chunks and merge them
header: allocated space keeps it size plus additional info in a header -> no size param when calling free()
free: freed region becomes new free-list-header, node next on old header
Best/Worst/First/Next Fit: Best/Worst expensive searching, First/Next cheap but external fragmentation
Segregated List: extra list for frequently requested objects
Buddy Allocation: when request: divide free mem by 2 until block fits requested size and another split makes block to small. "buddy" of this block has the same size. when block gets freed check if its buddy is also free. buddys adress differs by one bit -> easy adress determination, but: only power of 2 sized blocks -> internal fragmentation (request 6b, smallest fit 8b block)



chapter 18 Paging Introduction

paging: similar to segmentation, but now with fixed size chunks
page table: *per-process* data structur mapping virtual pages to physical pages. lies in os managed memory (4kb typical size)
VPN: virtual-page-number. ex: virtual adress 21:0b010101
PPN/PFN: Physical Page/Frame Number
PTE: Page-Table-Entry, holds translation and additiona bits
valid bit: marks unused pages invalid (i.e. when pro just started)
protection bit: determines read/write/execute acces on a page
present bit: indicates whether a page is in physical memory or on disk
dirty bit: indicating whether a page has been modified since it was brought into memory
reference bit: track whether a page has been accessed.
(accessed bit) good to see if a page is used regulary
supervisor bit: determines if user-mode processes can access the page
swap: swap parts of adress space to disk -> support adr space > ram
deamon: prozess ohne ein und ausgabe



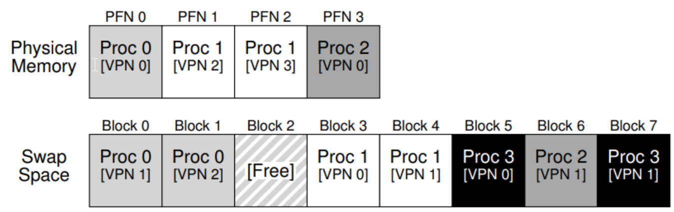
chapter 19 TLB

problem: adress translation via page table is slow -> Translation-Lookaside-Buffer(Cache)
TLB: Cache for **page table** entries. look for **VPN:** hit-> extract **PFN**, miss -> look in page table and write entry in **TLB** - retry instruction
spatial locality: data is tightly packed into pages, only acces to first element is a miss
temporal locality: keep recently used entries in cache (loop)
TLB Miss: CISC/Hardware: **PT** adress safed in **table base register**, "walk" page table, extract translation, update **TLB**
RISC/Software: hardware raises exception, pauses instruction stream, raise to kernel mode, jumps to **trap handler(**OS code). use privileged instruction to search entry, update **TLB and return from trap** (retry instruction that caused the **trap**)
TLB miss when accesing "miss handler code"? -> code on physical adress/
wired register: reserve TLB entries for OS
problem: how to make sure that a pro only use its own entires in the TLB?
flush: set all **valid bits** to zero. -> high cost when many **context switches** occur
ASID: adress space identifier field in the **TLB** to determine to which process a entry belongs
cache replacement: LRU(last-recently-used)/**random**
global bit: used to share pages between pros
TLB coverage: pro wants do acces more **pages** in a short time periode as the **TLB** can hold

Chapter 20 - Paging: Smaller Tables

problem: Page table is to big! even unused(invalid) **pages** are maped
bigger pages: Page size * 4 = **PT.**size/4, problem: **internal fragmentation**
hybrid: split **PT** into 3 segments (code/heap/stack) and hold **base** and **bounds** for each **PT** segment
problem: segmentation is not flexible/ **external fragmentation** because segments are not **Page**-sized
Multi-level-PT: chop **PT** into **Page** sized units. if entire **Page** of **PTE** is invalid dont allocate it.
page directory: track wether or not full **Page** of **PTEs** is valid and its **PFN=PPN**
less adress space used => smaller **PT,** grow **PT** by adding allocating another **Page** + set **PDE** on valid
but: translation on **TLB** miss is more expensive (**time-space trade-off)** + higher complexity
PDE: page-directory-entry with **valid bit** and **PFN**
valid bit: true if at least one **PTE** on this **Page** is valid
PDBR: Page directory base register
Vadress: num of **PTE** = adr-space-size / **page**-size **|** num of **PDE** = **PTE** / (**Page**-size / **PTE**-size)
more levels: if **PD** doesnt fit on one **Page** -> add another level and split **PD** into **Page**-sized chunks
Inverted PT: One **PT** with all *physical pages* that tells us which process is using this Page

Chapter 21 - Swapping



swapping: fit greater adress space in memory by swapping some **pages** to **disk**
present bit: indicates wether or not a **page** is present in memory or on **disk**
page fault/miss: pro tries to acces **page** that is not in memory but on **disk**
fault handler: block process. Use **PFN** in **PTE** to store the location of the page on disk.
OS loads page from disk to memory. **OS** updates **PFN** and **present bit.** retry instruction
OS may have to **page out** one page to make room for the new one
swap/page deamon: frees some pages if there are less than **low watermark** pages free available

$$\text{AMAT} = T_M + (P_{Miss} \cdot T_D)$$

- T_M Zugriffszeit auf Hauptspeicher
- P_{Miss} Wahrscheinlichkeit eine Seite nicht im Cache zu finden [0,0..1,0]
- T_D Zugriffszeit auf Platte/Disk

Chapter 22 - swapping policies

AMAT: average-memory-access-time
cold start miss: initial misses when an empty cache begins to fill up
capacity miss: cache ran out of space
conflict miss: "capacity miss" in one segment of a **set-associative** cache
FIFO: good for loops, but cant determine the importance of a block
Random: manchmal sehr gut(close to optimal), manchmal aber auch sehr schlecht(fast keine hits)
LRU/LFU: Least-frequently/recently-used problem: n sized memory, but n + 1 pages frequently used
array to determine use frequency of a page -> slow
used bit + clock algortihm: page used => used bit to 1 by hardware.
clock hand: point to one page. replacement occurs: if present bit = 1 set it to 0, go to next page
if present bit = 0 **page out**
dirty bit: modified pages h
prefetching: if page P is brought into memory page P + 1 will likely soon be accessed and should be brought into memory too
clustering/grouping: perform a single large write instead of m,ultiple smal ones
trashing: admission control run not all pros so that some can finish faster make room for new ones
out-of-memory-killer: kill memory intensive process

Chapter 26 - Concurrency

thread: a process with mutliple threads has more than one point of execution(multiple PC's)
-> switching between threads similar to content switch. but same AdressSpace
TCB: Thread control block to store the state of a thread (registers, PC etc)
PCB: Process control block to store the state of a pro (registers, PC, page table etc)
thread-local-storage: each thread has its own stack, it cant be accesed by other threads (normaly)
why use threads?: parallelism + use cpu on one thread while the other waits for I/O
race condition: two threads use the same variable -> read + write takes multiple assembly instructions
-> if thread gets interrupted the read value can be changed by other thread
atomic operations: execute a set of instructions together or dont execute them at all
critical section: is a piece of code that accesses a shared resource, usually a variable/data structure
inditerminate program: programm output is not deterministic
mutual exclusion: only allow one thread to enter critical section

chapter 27 - Thread API

pthread_create: to create a new thread. args: thread*, thread args(i.e. NULL), function* to start, arg* for function arguments

```
#include <pthread.h>
int
pthread_create(pthread_t        *thread,
          const pthread_attr_t *attr,
                void           *(*start_routine)(void*),
                void           *arg);
```

```
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

pthread_join: wait for a thread. args:
thread to wait for, void** for return values
*int pthread_join(pthread_t thread, void **value_ptr)*

pthread_create followed by pthread_join == procedure call

```
int main(int argc, char *argv[]) {
    pthread_t p;
    myret_t *rvals;
    myarg_t args = { 10, 20 };
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &rvals);
    printf("returned %d %d\n", rvals->x, rvals->y);
    free(rvals);
    return 0;
}
```

locks:
*pthread_mutex_init(pthread_mutex_t *)*: initialize a new lock(check return value for error)
*pthread_mutex_lock(pthread_mutex_t *)*: try to grab the lock till its succesfull
*pthread_mutex_unlock(pthread_mutex_t *)*: release the lock
*int pthread_mutex_trylock(pthread_mutex_t *):* returns failure if unable to grab lock
*int pthread_mutex_timedlock(pthread_mutex_t *,struct timespec)* returns failure if unable to grab lock or after timeout
*int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *):* sleep and "free" the lock (second argument) reaquire lock when getting waked
*int pthread_cond_signal(pthread_cond_t *):* wake a other thread whos waiting

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

chapter 28 - Locks

"ranking locks": correctness, performance, fairness (starvation)
Controlling Interrupts: simple but: trust issue, doesnt work on multiprocessor interrupts can become lost, slow
Simple Flags: flag=1=locked -> spin-wait. no mutual exclusion wastes cpu cycles
Hardware-support: "test-and-set/atomic exchange"
atomically read/test/write flag, **preemtive scheduler** needed to interrupt thread
mutual exclusion,deadlock-free,not bounded, no fairness guarantee,
good performance if numThreads ~ numCPUs

"Compare-andSwap"
similar to test and set

"Load-Linked and Store Conditional"
similar to atomic exchange, LL fetches value, SC stores new value only if no intervening store has taken place

"Fetch-and-add"
atomically increments variable and returns old value -> ticket lock possible
deadlock-and starvation-free

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Figure 28.7: **Ticket Locks**

petersons algorithm: test-and-set with 2 flags (flag[],turn) without atomic op
Just Yield, Baby: instead of spin wait give up cpu (running -> ready)
-> less wasted cpu cycles, more **context switches**
Queue: guard flag to "lock the lock", if lock free, acquire it,
if not add to queue, set guard to 0 and park. when unlocking give lock directly to waiting thread if there is one (unpark(threadID))/ setpark() to prevent race condition
Two-Phase-Locks: spin wait for a short amount of time, then yield

chapter 29 - Lock-based Concurrent Data Structures

threadsafe: multiple threads can safely and concurrently acces the data structure
perfectly scalable: 2 Threads do twice the work in the same time 1 threads needs
problem: just putting locks around critical sections allows only 1 thread at a time to acces structure ->not conc.
approximate(sloppy) counter: let each thread increment a local counter. once it reaches a **treshold** resett it and increment global counter by that ammount
hand-over-hand locking: dont lock whole list. each node has its own lock. traversing list by getting nexts->lock and releasing current->lock
queue: head and tail, enqueue at tail, dequeue at head
hash-table: scales very good. each "hash-bucket" is a linked list with a lock -> many concurrent ops possible

chapter 30 - condition variables

pthread_cond_t: type of a condition variable. "supports" wait() and signal()
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m): put calling thread to sleep and release lock
pthread_cond_signal(pthread_cond_t *c): wake up one thread waiting on condition variable 'c'
pthread_cond_broadcast(pthread_cond_t *c): like signal but waking all threads waiting on 'c'
producer-consumer/bounded buffer problem:
if instead of while: thread gets woken, but before int ran another threads clears buffer -> cv just a hint
one condition variable: consumer finds empty buffer, calls signal -> another consumer could be woken, finds empty buffer, calls wait -> nobody wakes producer -> deadlock
Mesa vs Hoare Semantics: Hoare immediatly schedules woken thread, mesa puts it to ready so it has to re-check if condition is still true

```
struct semaphore {
    int value;
    queueType list;
    }

sem_wait(semaphore *S) {
S->value--;
if (S->value < 0) {
     add this task
        to S->list;
     block();
 }
}

sem_post(semaphore *S) {
S->value++;
if (S->value <= 0) {
     remove a task P
 from S->list;
     wakeup(P);
  }
}
```

chapter 31 semaphores

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;
```

semaphore: object with an integer value that we can manipulate with two routines
sem_wait(): decrement sem-value by one, wait if it is negative
sem_post(): increment sem-value by one, if on or more threads are waiting wake one of them
binary semaphores: value is initialized with one. works like a lock
ordering primitive: init value with 0. works like condition variable. (parent(wait) waiting for child(post)):

1. case: parent runs, decrements value to -1, waits -> child runs increments value to 0, wakes parent
2. case child runs increments value to 1, nobody to wake -> parent runs decrements value to 0, value not negative -> return and go on

throttling: init sem->value to a treshold x. now x threads can acquire the "lock" at the same time

Bounded Buffer/Producer-Consumer Problem

Mutex Scope is to big -> deadlock possible, solution: mutex just around put/get

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line P0 (NEW LINE)
        sem_wait(&empty);        // Line P1
        put(i);                  // Line P2
        sem_post(&full);         // Line P3
        sem_post(&mutex);        // Line P4 (NEW LINE)
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line C0 (NEW LINE)
        sem_wait(&full);         // Line C1
        int tmp = get();         // Line C2
        sem_post(&empty);        // Line C3
        sem_post(&mutex);        // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

```
void vector_add(vector_t *v_dst, vector_t *v_src) {
    Pthread_mutex_lock(&v_dst->lock);
    Pthread_mutex_lock(&v_src->lock);
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
        v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}
```

chapter 32 Concurrency Bugs

Atomicity violation: no locks used
Order violation: use condition variable if order is important
Deadlocks/Tödliche Umarmung: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does
v2.AddAll(v1): if v1 and v2 need to be locked it matters if v1 or v2 is the parameter (lock order)
prevention via scheduling: possible but expensive
deadlock detector: detect deadlock and reboot
livelock: 2 people in a narrow corridor move aside (but to the same side) to be polite over and over again -> no on can go on walking
Bankers algorithm: If a resource is going to be claimed the system is checked, if by this claim a deadlock is possible; no deadlock possible -> system safe, allocate; deadlock possible -> system unsafe

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

Solutions:

atomic instruction instead of locking

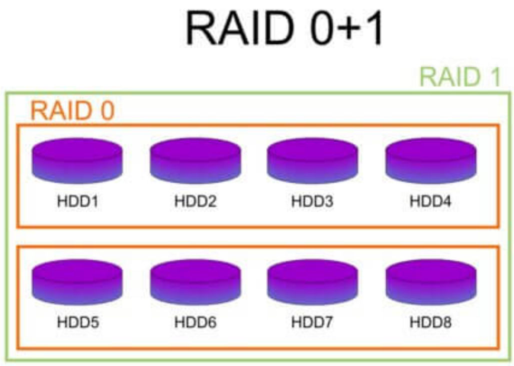
acquire all locks at once (global lock around local locks)

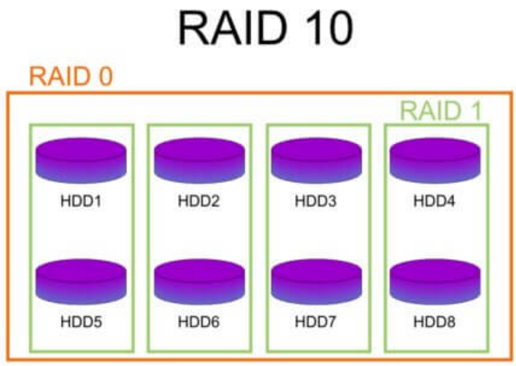
grab first lock, realease following locks if canot acquire all of them (trylock)

Total ordering of locks

chapter 10 Multiprocessor Scheduling

SQMS: single queue multiprocessor scheduling: one queue, if core is idle it gets next job on the queue -> joby run on different cores
MQMS: multi queue multipro scheduling: as many queues as cores, each core runs next job on its queue
cache coherence: pro loads value, edit it, saved in cache -> pro gets **migrated** to other core, reads value again but dont find it in cache -> fetches old value from memory (write-back-cache)
bus snooping: observe bus: if update to memory that is also in cache happens update or invalidate said entry in cache
Cache affinity: Thread/Pro builds up state in the cache of cpu -> switching to another core loses the state
Load Balancer: Keep all CPUs busy (by "stealing" jobs from busy core)
Linux Schedulers:
O(1): multi queue, priority based (similar to MLFQ)
CFS: multi queue, proportional share approach (like stride scheduling)
BFS: single queue, proportional share, Earliest Eligible Virtual Deadline First (EEVDF)
Load Balancer Details:
1.Lock all queues (ordering matters)
2.Find_busiest_queue() of other cpus
at least 25% more
3.Loop
Select highest priorty task from the expired queue
Check if selection is:
- not cache hot
- not pinned to cpu by processor affinity
exit loop or start over with second highest priorty task ...
4. Move selection to own cpu
5. Repeat 2-4, until all queues are balances
6. Unlock all queues





RAID Comparison: A Summary

N : the number of disks
D : the time that a request to a single disk take

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N	N/2	N-1	N-1
Reliability	0	1 (for sure) N/2 (if lucky)	1	1
Throughput				
Sequential Read	N · S	½N · S	(N-1) · S	(N-1) · S
Sequential Write	N · S	½N · S	(N-1) · S	(N-1) · S
Random Read	N · R	N · R	(N-1) · R	N · R
Random Write	N · R	½N · R	½R	¼N · R
Latency				
Read	D	D	D	D
Write	D	D	2D	2D

RAID Capacity, Reliability, and Performance