Shrinithi Narayanan

*What is an OS

- does not have a "main()" and doesn't really exit
- not like a normal program, more like a superative library
- called by programs and hardware
- bootstrap sets OS up, then used/called however
- acts as an intermediary system between app program + hardware
  e.g. who has permissions to access certain files
- provides an environment in which to run programs, provides services

HW ↔ OS ↔ Apps ↔ User

*Goals of an OS

- easy, consistent to use computer (better than barebones)
- efficient use of resources; e.g. abstraction of space and time ("simaltaneous" processes)

*What things from vendor

- compiler? window system? web browser? X NONE (by our definition)

INSTEAD: Everything that was in hardware "kernel" mode ("core")
(like UNIX "root" or Windows "admin")

- other essentials? init processes?

*OS does 2 things

1. Create abstractions
   - disk → files, directories
   - CPUs → processes, context-switching
   - memory → address spaces, virtual memory
2. Manages resources
   - hardware and software
   - increase hardware utilization = $\frac{\text{useful time}}{\text{total time}}$

*How does the OS do this

fool the user, fool the hardware. (hardware doesn't need to be aware of OS abstractions)

*History of OS

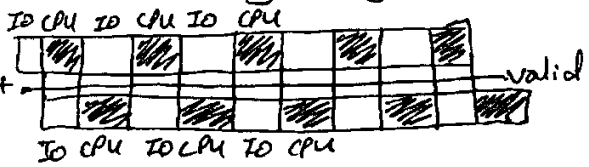
- Early history = no OS, just toggled-in instructions, time slots (punch-cards!) inefficient 1940's - 1950's
- Early batch systems - non interactive, patching many programs together 1950's - 1960's
  - a simple Resident Monitor — run program
  - a job control language
  - memory loads/at boot time into memory

Multiprogramming - multiple "processes"
memory split among the many programs (for each process)
context switch only when needed (no need for extra overhead)

→ 1960's - present

Shrini Narayan

efficient context switching usage of CPU



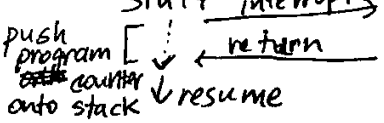
Multix! Unix!

timesharing - interactive multiprogramming 1960's ... 1970's — present
- context switching fast enough to simulate "timesharing", move frequent (100+x frequenter) Batch: turn-around time → time to completion
Interactive: response time → time to first response

Hardware Support for OS

problem: IO is slow
busy waiting: ~~while~~ while (no char ~~yet~~?) "necessary" but unproductive
how to know when to switch back here?

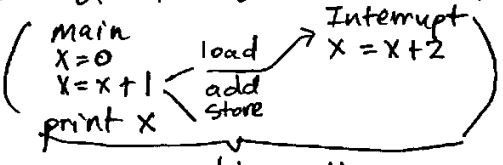
new hardware feature: interrupts
definition: an interrupt is an external signal (outside CPU) to the CPU (is async) Main program Interrupt handler



e.g. take char from keyboard hardware
return instruction
asynchronous
subroutine call

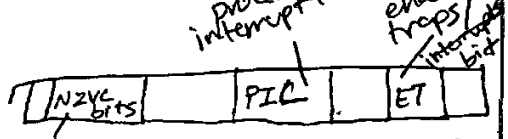
the interrupt vector table
in memory, an array of pointers to functions/procedures
on interrupt, external hardware gives interrupt type
location? (hardware needs to know where) → new special register! ™ =
definition: the vector base register holds the location of the address of the interrupt vector table

vector base register
vector here means the ptrs to the other functions



X can be 1 or 3
we need better interrupt handling

hardware synchronization for interrupts
two mechanisms: disable + enable interrupts
example: SPARC processer storage register (it disables all interrupts)



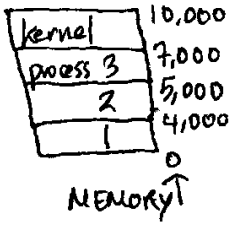
to indicate conditions like overflow

when "disable" pending interrupts are remembered: but only one of each type of interrupt
interrupt properties: new interrupt allowed only if it's a greater priority interrupt than current level priorities for interrupts

Shrinithi Narayanan 3

Protection

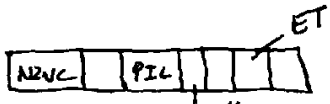
ist be built into hardware (software too slow, insecure)



- Two new special registers
  - BASE: first legal address (process 3: 5,000)
  - LIMIT: number of bytes allowed (process 3: 2,000)
- address X is legal iff $0 \leq X \leq$ LIMIT
  - ↳ If legal, hardware then accesses x+BASE
  - ↳ Actual BASE is hidden from process (for security)

Dual-Mode Execution

Example: SPAR, ~~PSR~~PSR



"supervisor" bit (kernel mode on/off)

only allow changes when in kernel mode

"~~privileged~~ privileged instructions": only usable in kernel/supervisor mode

- e.g. changing ~~BASE~~ or LIMIT, changing vector base (interrupts), changing PIL, S, ET, perform IO, halt instruction

Definition: ~~An~~ An interrupt is an async signal from external source

Definition: ~~An~~ An exception is an error in current instruction execution

Definition: A trap is a special instruction used by user mode to request service from the kernel

all three used operations are similar:

usage of interrupt table for all 3

- however, only interrupts can be disabled (not ~~traps~~ traps)
- all 3 goes into kernel mode
- all push current PSR and PC onto stack (for returning)
  - PSR: processor status register
  - PC: program counter
- *these are the only 3 ways into kernel mode*

- these 3 act as the only, controlled ways into kernel mode
- the clock interrupt: generates periodic interrupts so that kernel mode would have some consistent control (eg per 10 ms)
  - allows OS to regain control when needed eg for context-switching
  - can be used to count time, down time-outs, CPU usage

Synchronization/Race Issues

Def: process is a program in execution; has own address space, but can be "multi-threaded"

- threads (also called "lightweight processes") can share same address space

race condition: different "processes" sharing data; outcome dependent on order of events — Shrini Narayan

the outcome/order is not determined by the program

- maybe can solve with disabling/enabling interrupts, but problematic and inefficient (messes up clock, only works w/ single processor)

- high level tool for process synchronization

monitor foo { define shared vars: int x
entry inc_one() x = x+1;
entry inc_two() x = x+2; }

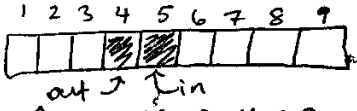
essentially a lock mechanism — when entry starts, acquire mutual exclusion over whole monitor.

- sync enforced by compiler/runtime
- only 1 process active at a time in monitor
- can only call entries from outside
- ~~when~~ upon return, release mutual exclusion (a "lock")

History of monitors

Three different types, Per Brinch Hansen 1973, CAR Hoare 1974, Mesa language 1974

The Bounded Buffer Problem



circular buffer w/ in, out ptr counter as well

a buffer of N items, a producer, a consumer

producer: while (1) { produce item; place item in buffer }

consumer: while (1) { take item from buffer; consume item }

need a way to view buffer as full or not w/o it changing

- new variable type: no value type, only used inside monitor (not handlable by monitors)

two operations: X wait - blocks current process until...

(different rules for how to handle X signal) X signal - wakes up one process if there is one waiting

- Hoare monitors (usually followed): P is blocked, Q does X signal (signal and wait)

Q is blocked, transfer automatically to P

Q continues ~~to~~ run later after P leaves monitor or P waits on some other condition

- Mesa monitors (signal and continue)

Q continues to run; P runs later after Q leaves or wait on condition var

- Brinch-Hansen monitors (signal and return)

- two ways to leave: return, or signal and return

→

(s) to Bounded Buffer Problem (using condition vars under Hoare monitor)

```
tor bounded { struct item buf[N]; int in=0, out=0, count=0; condition empty, full;
```

ssuming e producer+consumer problems can arise with multiple producers/consumers (who gets woken up?)

```
entry add_item(struct item *data) {
while (count == N) {full.wait}
buff[in] = *data ; in=(in+1) % N;
count++; empty.signal; }

entry remove_item(struct item *data)
while (count == 0) {empty.wait}
*data = buf[out] ; out=(out+1) % N;
count--; full.signal; }
```

Starvation: infinitely unable to get resources while others keep succeeding

The Readers-and-Writers Problem

- one shared object, any # of 2 kinds of processes: readers and writers
  - readers: read-only
  - writers: read and write
- any # of readers active at once; if reader active, no writer allowed and vice-versa

Mesa Monitor Solution

```
Monitor RW { int num_reader=0, num_writer=0; condition reader, writer;

entry begin_read() {
  while (num_writes ≠0) reader.wait;
  num_reader++; }
                ← can add reader.signal
entry end_read() {
  num_reader--;
  if (num_reader==0) writer.signal; }

entry begin_write() {
  while (num_writer≠0 or
    num_reader ≠0) writer.wait;
    num_writer++; }
entry end_write() {
    num_writer--;
    reader.signal; writer.signal; }
```

Hoare Monitor Solution

introduce accounting of bypassing to prevent starvation (of writers)

```
Monitor RW { int num_readers, num_writers
             condition reader, writer
             int waiting_readers, waiting_writers
             int bypassed=0

#define bypass_limit 1000000

entry begin_read() {
  if (num_writers>0 || bypassed ≥ bypassed_limit)
      waiting_readers++;
      reader.wait;
      waiting_readers--;
  num_readers++;
  if (waiting_writers>0) bypassed++;
  if (waiting_readers>0 and bypassed < bypassed limit) reader.signal

entry end_read() {
  num_readers--;
  if (num_readers==0 and waiting_writers>0)
      writers.signal

entry begin_write() {
  if (num_readers>0, num_writers>0)
      waiting_writers++;
      writer.wait;
      waiting_writer--;
      bypassed=0
  }
  num_writers++;

entry end_write() {
  num_writers--;
  if (waiting_readers>0)
      reader.signal
  else if (waiting_writers>0)
      writer.signal
```

Dining Philosopher's Problem

Shrini Narayan

each philosopher can only use a fork to the left or right
can only pick up 1 fork at a time; must hold both forks to eat
must be possible for 2 to eat at once — Djkstra 1965
starvation, deadlock, livelock possible: e.g. each picks up left fork → deadlock

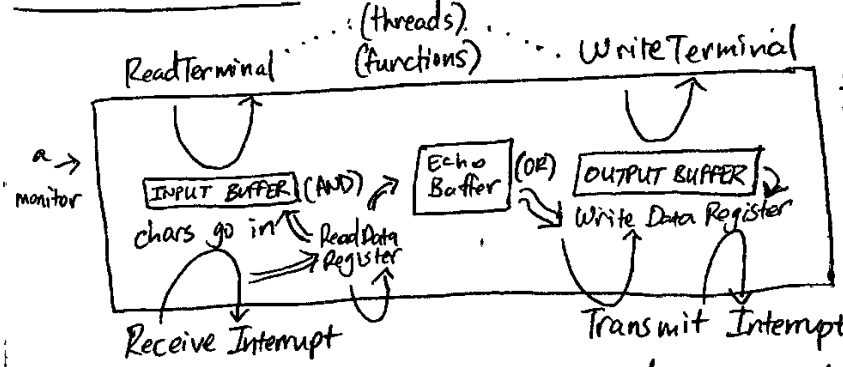
```
while (1) {
  [think]
  eat
```

Hoare Monitor

```
monitor dining { int state[5] = "thinking" THINKING, EATING, HUNGRY }
                 condition philcond[5]

entry pickup_forks(int i) {
    state[i] = HUNGRY
    test_forks(i)
    if (state[i] ≠ EATING)
      philcond[i].wait
}

entry putdown_forks(int i) {
    state[i] = THINKING
    test_forks((i+1)%5)
    test_forks((i+4)%5)
}

test_forks(i) {
  if (state[(i+4)%5 ≠ EATING and
    state[(i+1)%5] ≠ EATING and
    state[i] == HUNGRY) {
     state[i] = EATING
     philcond[i].signal
}}
```

Lab 1 Review

Device driver for a set of terminals written as Mesa monitor



(app/terminal) app interacts w/terminal
(driver) terminal
↑ chars in
(hardware) keyboard/device

Flag to keep track of whether a ReadData or WriteData has happened
keep cycle of transmit interrupt → write data register

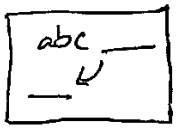
Input

Keyboard: ReceiveInterrupt

- "\r" or "\n" (^M or ^J)
- "\b" or "\177" backspace delete

input buffer	echo buffer
"\n"	"\r\n"
non-empty last line • remove last char else, do nothing	non-empty last line • "\b \b" else, optional \7

terminal



abc_ → abc → ab_ → cb_ (∝ bell)

"helloworld" or "worldhello"

must preserve "scripts" (e.g. WriteT("hello") and WriteT("world") ⇒ "worldhello"
block until everything written or everything read
external functions for monitors (not "static")

script typescript cat - √ typescript

```
extern foo(arg) {
    Declare_Monitor_Entry_procedure()
}
cond_id_t X;
X = condcreate();
condsignal(X);
```

⟺

```
entry foo(arg) {
}
condition X;
X.signal;
```

...ores - new variable type, like int, but after initializing, 2 operations for semaphores

P(S) => { while (S≤0) do nothing; } atomic except (allow context switch when waiting) "down"
S = S - 1

V(S) => S = S+1 ← atomically "up"

careful use required (can deadlock)

Example use: Bounded Buffer Problem

Semaphore mut_ex = 1, empty = N, full = 0 ← need to for limiting both the producer and consumer
so, also need mutex to keep "empty" and "full" synced

producer: while (1) {
  produce item
  P(empty)
  P(mutex)
  put item in buffer
  V(mutex)
  V(full)
}

consumer: while (1) {
  P(full)
  P(mutex)
  take item from buffer
  V(mutex)
  V(empty)
}

mutex is "binary" (0 or 1)
empty, full is "general" (0 to N)
defined by usage

Readers and Writers Solution

int readcount = 0;
semaphore mutex = 1, writing = 1

writer: while (1) {
  P(writing)
  write
  V(writing)
}

reader: while (1) {
  P(mutex)
  readcount++
  if (readcount == 1)
    P(writing) ← first reader "locks/frees" writers
  V(mutex)
  read
  P(mutex)
  readcount--
  if (readcount == 0)
    V(writing) ← last reader
  V(mutex)
}

still has same problem of starving writers; add another bypass or semaphore

Dining Philosophers Solution

semaphore fork[5] = 1 ← table = 4

philosopher (i) {
  while (i) {
    think...
    P(table)
    P(fork[i])
    P(fork[(i+4)%5])
    eat...
    V(fork[(i+4)%5])
    V(fork[i])
    V(table)
  }
}

deadlock is possible if everyone picks up fork at same time; adding a "table" semaphore avoids deadlock: a FIFO semaphore eliminates starvation

assumption of fairness in "table" is key

adapt idea for our (Hoare) monitor, again assume fairness

monitor dining { int table = 0; condition seat; bool avail[5] = 1; condition fork[5]

entry pickup_forks(int i) {
  if (table == 4) seat.wait
  table++
  if (avail[i] == 0) fork[i].wait
  avail[i] = 0
  if (avail[(i+4)%5] == 0) fork[(i+4)%5].wait
  avail[(i+4)%5] = 0;

putdown_forks ⟹

```
entry putdown_forks(int i) {
    avail[i] = 1
    fork[i].signal
    avail[(i+4)%5] = 1
    fork[(i+4)%5].signal
    table--
    seat.signal
}
```

The "General Critical Section Problem" Shrin[redacted] Narayan[redacted]

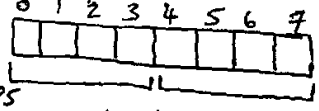
Assumptions: all processes execute critical section at nonzero speeds; no assertions on relative speeds of processes

Requirements: mutual exclusion: only one process in critical section at a time

progress: if some process wants to enter critical section and no process in section, then only processes in enter or exit section influence choice of next process, and choice cannot be indefinitely postponed

bounded waiting: if process wants to enter critical section and has begun entry section, you can't be skipped indefinately

Implementing Synchronization (for process 0 and 1 for now)



memory spaces

- Shared variables required
  - int x automatic assignment to 4 bytes
  - int *y pointer to 4 byte space
  - char z[4] doesn't know if z
  - y = (int*)z is multiple of 4
  - *y = 5 ← store may thus be problematic
- using 1 shared var for synchronization
  - int flag = 0
  - entry while(flag==0) flag=1
  - exit flag=0
  - ⇒ no mutual exclusion order of these 2 ops
  - int flag = 1
  - enter while (flag ≠ 0) ← deadlock
  - exit flag = 0
  - process-i
  - int turn = 0
  - entry while (turn ≠ i) ; lack of progress if process doesn't take turn
  - exit turn = 1 - i
- using 2 shared vars for synchronization
  - int flag[2] = 0, 0
  - enter while (flag[i-1]) ⇒ no mutual exclusion
    - flag[i] = 1
  - exit flag[i] = 1
  - flag[i] = 1
  - while (flag[i-1])
  - flag[i] = 0 ⇒ deadlock
  - enter: flag[i] = 1
    - while (flag[i-1])
    - flag[i] = 0
    - wait a little
    - flag[i] = 1
  - exit: flag[i] = 0
  - violates bounded wait (perfect sync, low probability)
- using 3 shared vars for synchronization
  - Dekker's algorithm (1965)
  - int flag[2] = 0, turn = 0
  - (combo of turn and flags)
  - See book for full algo
  - Peterson's Algorithm (1981)
  - enter: int flag[i] = 1, turn = 1 - i
  - while (flag[1-i] and turn == 1 - i) ← critical point
  - exit: flag[i] = 0
  - solution to critical section problem for 2 processes
  - but not very efficient busy waiting + only 2 process

rt's Bakery Algorithm (1974) for multiple N processes

dispensing machine for the processes

if choosing[N] = 0, number[N] = 0

```
enter { choosing[i] = 1
        /* number[i] = max(number[...]) + 1 */
        int x, temp = 0
        for (x = 0; x < N; x++) {
            if (number[x] > temp) temp = number[x]
        number[i] = temp + 1
        choosing[i] = 0
        for (x = 0; x < N; x++) {
            while (choosing[x]) {;}
            while (number[x] ≠ 0 && (number[x] < number[i] ||
                   (number[x] == number[i] && x < i)))
                {;}
      }
exit  { number[i] = 0 }
```

choose a number larger than the max (this is the ticket number)

busy waiting efficiency

no process can have advantage over some process more than once

Hardware Support

test and set instructions

```
test_and_set x   } atomic
    temp = x
    x = 1
    return temp
```

```
x = 0
enter while (test_and_set x)
          ;
exit   x = 0
```

one var solution w/ hardware support, but may still have unbounded waiting

acts as a "lock"

```
swap a, b
    temp = a   } atomic
    a = b
    b = temp
```

```
lock = 0
local key = 1
while (key)
    swap(lock, key)
lock = 0
```

hardware supported

```
exit { local int x
       x = (i + 1) % N
while (x ≠ i and waiting[x] = false)
       x = (x + 1) % N
if (x == i) lock = 0
else waiting[x] = false
```

sequential loop thru buffer prevents starvation

Using hardware to reduce starvation

```
int waiting[N] = false; int lock = 0
entry {local int key
enter   waiting[i] = true
        key = 1
        while (waiting[i] and key)
            key = test_and_set(lock)
        waiting[i] = false
```

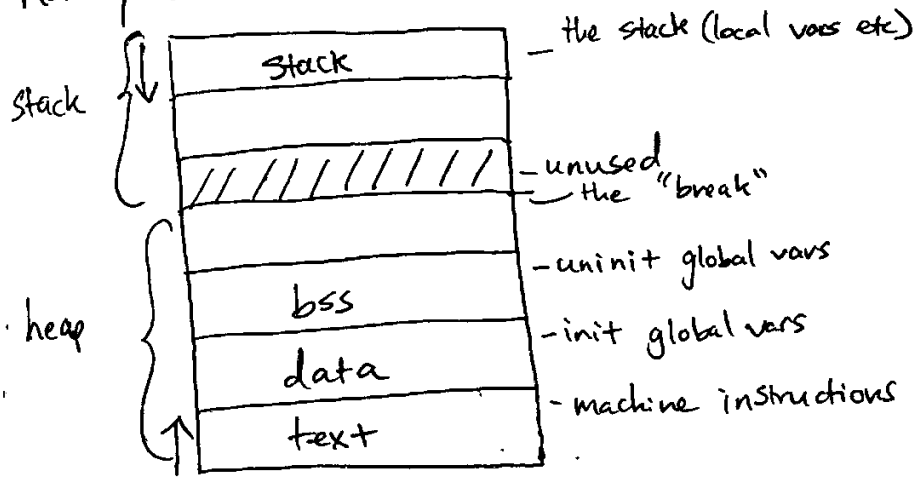
busy loop, to avoid use below

```
if (uniprocessor || lock holder known to hold a "long time")
    ask OS for context switch
else, for (i = 0; i < large # and still can't actually lock; i++)
    if (still can't acquire lock)
        ask OS for context switch
```

Processes, Memory Management

Shrini Narayan

Memory structure




```
int x=1
int y[1000]
main(...) {
  int a = 2;
  int b;
  a=b;
  proc();
```

register values: must be stored somewhere even when process not used
per process OS data structure: process control block (PCB) (track when not being used). PCB holds saved reg. values, pid, user id, uid, group id (gid), accounting info, lots of pointers!

have ready and active queues of PCBs (one for blocked?) circular linked list?

ready → → → ... pop and add for context switching

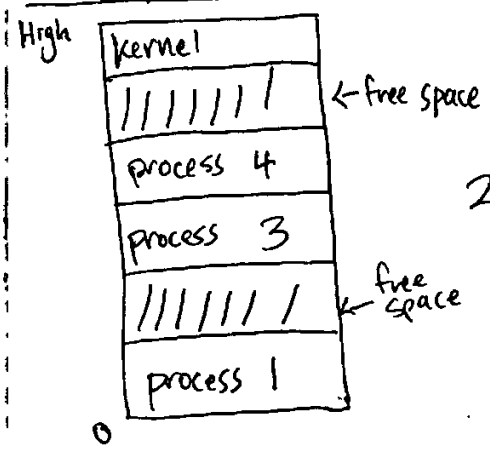
active → → → ... each square = PCB

blocked → → → ... ← maybe seperate queue per reason for being blocked

the "idle" process, X HALT
JMP X ← saves power before blocking (interrupt wakes up)
(when ready queue empty)

active PCB ≠ head of ready queue

Memory Management

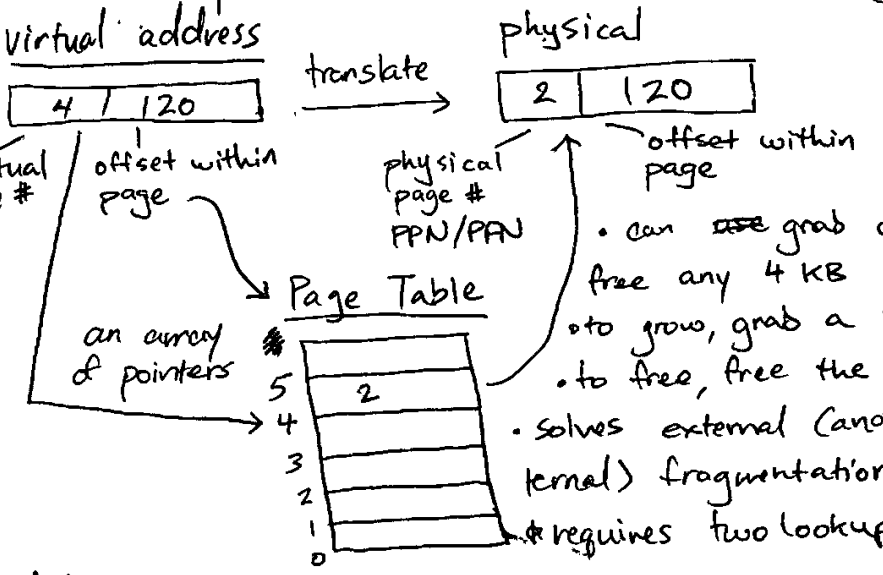


- "fix" by compaction (copy process memories down to free chunks) ← inefficient
- to allow process growth, must push for space, but over time, may have unused space in each process space

2 problems:
external fragmentation: enough free space for new alloc, but not enough contiguous
internal fragmentation: "trapped" memory inside allocated chunks left unused

→

divide memory into fixed sized chunks, pages (eg 4kB) Srinithi Narayanan 11

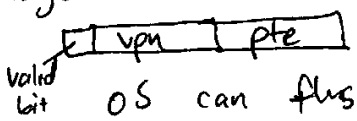


- can ~~use~~ grab and use and free any 4 kB amount of memory
- to grow, grab a page
- to free, free the pages
- solves external (and mostly also internal) fragmentation
- requires two lookups of memory per operation

the page table storage

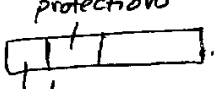
- special hardware registers (but table can get too large) (PTBR)
- in memory: then, tell hardware location with page table base reg and page table limit register (PTLR). PTBR - physical address of PT array; PTLR - # of entries in PT (physical addressos)
- after one initial physical address read, OS initializes PT, and virtual addresses are used.

Hardware Cache of Page Table Entries (the TLB)
- high hit rate = less need to fetch something from memory.
- managed almost entirely by hardware (lookaside to TLB before mem)



TLB entry; OS flushes TLB on Context switch
OS can flush single entry of TLB (if present)

Page Protection

PTE 

valid bit flag, 0 = not allowed access
also disallow PTE's out of range of PT range (if VPN ≥ PTLR, disallow)
protection bits: read/write/exec permissions for kernel + user (valid≠0)

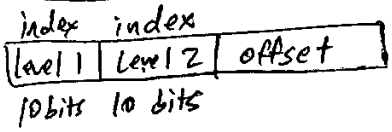
Kernel: where is it?

> own seperate address space? requires context switching to get in/out (expensive!)
problem: how to access user memory? →

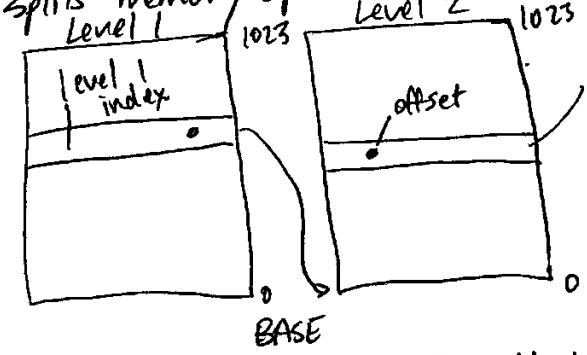
Shrinith Narayana 12

> in all process address space
problem of differing page numbers per process
problems w/ storing data and context switching
problem w/ interrupt vectors jumping to wrong spot
> clever use of virtual address space
use different page tables for each region (PTR0 and PTR1)
only modify PTR0 during context switch (be careful with protections)
kernel stack used (delicately placed in user process region 0)
synchronize virtual address space w/ physical address
that is, vpn <=> pfn, before enabling VM
build kernel heap at start (IVT, PTR1, stack pointer), map PTE's correctly, then enable VM

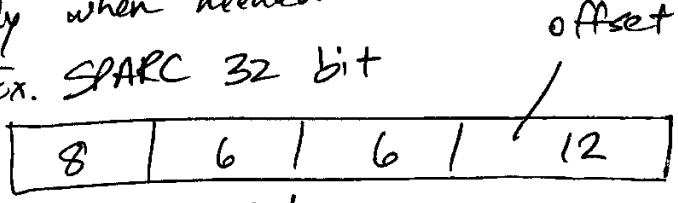
Tree structured (hierarchical) page table



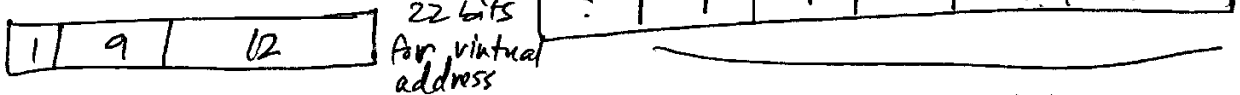
VR 2^{10} level indices a "tree" structure w/ 2 layers
$4\,KB \cdot 2^{10} = 4\,MB$
splits memory space to independent 4 KB chunks
saves space
level 2 usually sparse
flexible: filling chunks (1 page size) only when needed



Ex. SPARC 32 bit



Homework Lab 2 - RCS 421 64 bit Intel



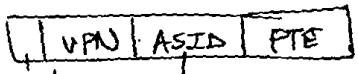
selects 1 of 2 registers, ~~each~~ pointing to a 4KB level 2 table.
level 1 isn't an array in this case, just registers
when loading program, one must place PC, SP in kernel stack
context switching: may be problematic since kernel stack disappears, instead repoint to shared kernel stack in region # of kernel.

Shrinithi Narayanan

13

have address space ID register (ASID) for each address space with TLB entries in TLB

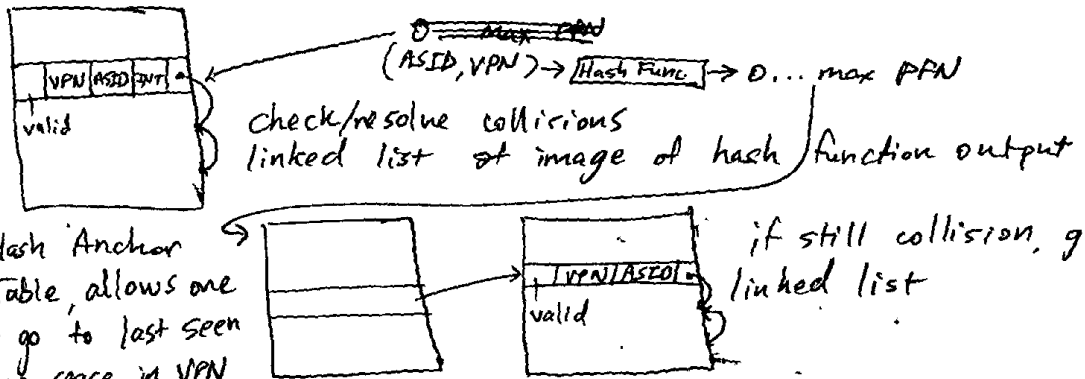
TLB entry



loaded w/ TLB entry, checked when looking up in TLB

for context switch prot

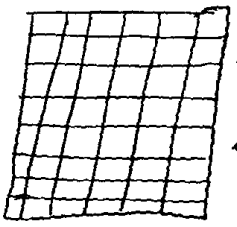
Inverted Page Table - only one per entire system



Hashed Page Table

all bits in one row?

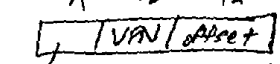
VPN, ~~AS~~ VSID
↳ hash func → PTE



in PTE, VSID + VPN check for actual match

prot, valid, pfn bits as usual

alternate ~~space~~ hash for more space for over-flowing bits.



segment → 15 virtual seg ID

segment regs, change w/ every context switch

reg

storage descriptor #1
- size + phys addr of table

context switch simple, entries in one compact place

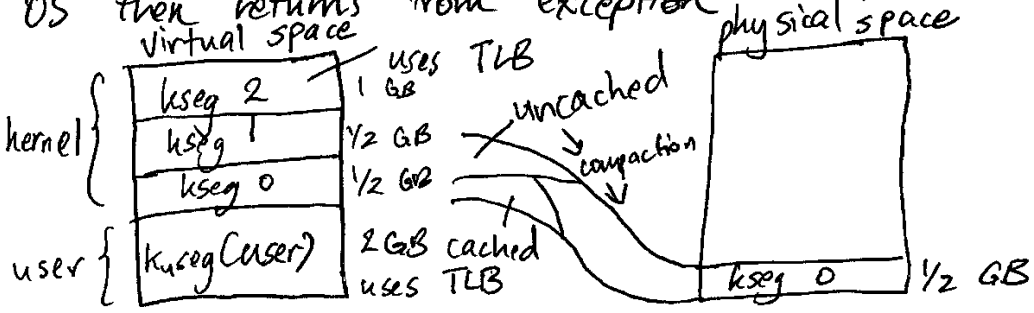
Recap: these are methods where hardware knows page format, TLB, complex structures, can we make the hardware really simple instead?

→

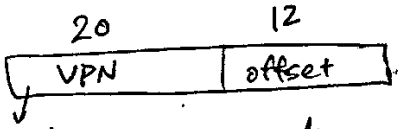
Software Managed TLB

Shrinit Narayanan 14

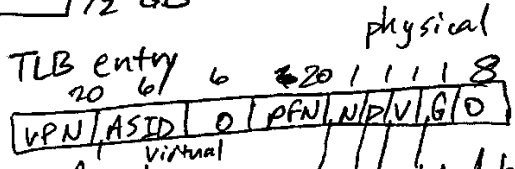
If entry not in TLB, HW causes TLB miss "exception" (HW not aware of page table structure)
- OS finds (or makes) needed TLB entry, loads into TCB instead of HW,
OS then returns from exception



virtual address



TLB entry



HW searches TLB for TLB valid == 1 and TLB vpn = addr_vpn

0 = kuseg
100 = kseg0
101 = kseg1
11 = kseg2

reading bits exhibits region of address

for prot (virtual)

not cacheable

dirty bit

valid bit

global bit

ASID = ASID_reg or TLB_G == 1

4 CPU registers

TLB entry LO - holds physical stuff for TLB entry (writeable)

TLB entry HI - holds virtual stuff

readibility is whether the page is in TLB

accessible from all ASIDs

index - "probe result bit", 6 bits - (TLB subscript)

random - "random number", 6 bits — counts from 8 → 64 predictable, but rapid counting

entries 1-7 safe from overwrite

4 instructions

TLBR - read the indexth entry

TLBP - probe TLB for entry matching TLB entry HI
(Hardware gets probe result = 1 on success and fills in TLB entry LO

TLBWI - write into TLB indexth entry from TLB entry HI and LO

TLBWR - write into randomth entry (not actually "random", but close

only overwrite entries w/ lock bit = 0 enough

64 bit SPARC

"TLB Data-In" register = TLB entry LO and HI

a TLB entry has a "lock" bit (writes down the entry in TLB: "locked"

removing/adding cached page tables applies to hashed page table

hashed page table vs. software TLB

Shrinithi Narayanan

ntation

native to paging

gment: a contiguous chunk of memory w/hardware enforced beginning + size

address = (segment #, offset in segment) ← wraps around to 0, doesn't

4 segment registers (since 86 386 FS) affect segment #

CS = code segment register (IPL holds offset)

DS = data segment register (instruction gives only offset)

SS = stack segment register (stack pointer holds offset)

ES = "extra" segment register Override prefix to change default segment register to use format instruction)

Segment Table

valid 0	prot	segment size	phys base adder

↳ PBA + offset ⇒ addr

2 tables – local process + global kernel

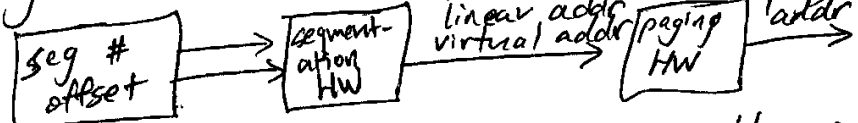
Local table Descriptor Register (LTDR) → phys addr #, num entries in table

Global table descriptor register (GTDR)

Segment register contains seg # = subscript in table and table indicator ~~(0=local, 1=global)~~ (0=global, 1=local)

segment register has: segment descriptor code regs = copy of that segment table entry

Segmentation w/paging CR3 = phys addr page table (not really used in C language)



to turn off segmentation, set table entries to 0, seg registers to 0, limit to MAX (base)

Used sometimes, for ex, for perror shared variables that appear differently for each thread

↳ ES register

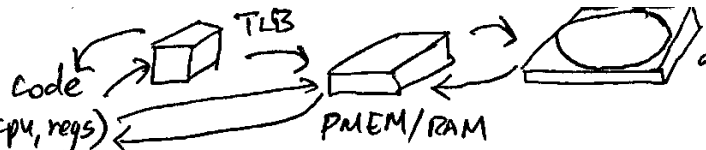
MORE ABSTRACTION!

Chapter 8: everything in VR space is in PR space (abstract) (physical)

Chapter 9: abstraction virtual space VR ⊆ PR

may be larger than pmem, physical pages become a cache of virtual pages, other virtual pages are on disk ("backing store")

Demand Paging



Shrinith
Narayan
16

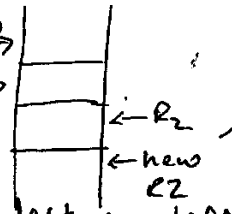
abstract
virtual space may be larger than pmem
physical pages become a cache of virtual pages
other virtual pages are on disk ("backing store")
- assumes: temporal locality - likely to access same piece of memory again in near future
spatial locality - likely to access same memory nearby in near future (Principle of Locality)

on miss ↓ fetch pg from disk, if no free phys page, "evict" one ↑ victims!

Thus, cache pages when they are accessed

- hardware requirement: all CPU instructions must be "restartable" being able to remember and do instructions so to undo instructions after, eg returning from an exception handler

example: mov R1 +, -R2
src dest



page exception,
↳ after returning, can't undo pointer movement

copy buffer to buffer
ex. move 5 srclen, src, file, destlen, dest ← differing buffer sizes = partial copying must be undone

page table entry: hit or miss

pte valid = 0 for all virtual pages not in physical mem, if pte valid == 0, hardware generates exception. if because of demand paging, call a "page fault" - OS finds free physical page on free list, or evict some other page (victim) from phys mem to evaluate locality of PTE's use

extra HW support
in PTE, referenced bit: set by HW when HW used its PTE
dirty (modified) bit: set by HW when HW uses that PTE for a while

→

Replacement Algorithms (victim selection)

Shrinithi Narayanan

1: random (bad idea)] both ignore principal of locality
constant

2: minimum (optimal ~~optimal~~) "page reference string" 2 1 4 2 3 0 4 2 5 4 (predict future... pagefault 17)
victim = page whose next use is farthest into future 0/2 1/X0 2/4 3/2
depends on size of phys memory and quality of prediction

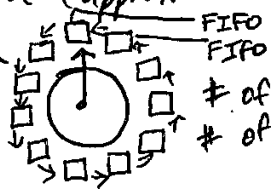
3: FIFO victim = "oldest" page (time of insertion)
eg 2 1 4 2 3 0 4 2 5 4 — 0/2① 1/X② 2/4③ 3/2④ 3/2⑤ 3/2⑥ 3/2⑦ 3/2⑧

4: LRU victim = last recently-used (time of access) queue over time
eg 2 1 4 2 3 0 4 2 5 4 — 0/2 1/X0 2/4 3/2 3/2 3/2 3/2 3/2 3/2 3/2
"pages referenced recently tend to be referenced again"

2	1	4	2	3	4	2	5	4	
	2	1	4	2	0	4	2	5	
		2	1	4	3	0	4	2	
			1	2	3	0	0		

Implementing LRU (approximation).

• Clock algorithm




```
while (1) {
    advance hand (clockwise)
    if (PTE[hand].ref == 0) break;
    else clear PTE[hand].ref (set to 0)
}  hand points to victim!
```

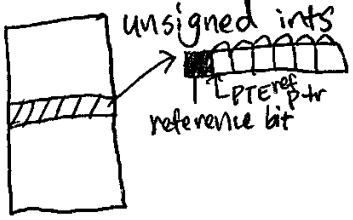
gives each page a "second chance" to get used again within a cycle, otherwise, victimized
remove victim from linked list, add new PTE to head of FIFO
Can also try using the dirty bit, but policy is subjective...

• enhanced second-chance algorithm
favor clean over dirty in victim selection

order of victim pref	ref	dirty
1	0	0
2	0	1
3	1	0
4	1	1

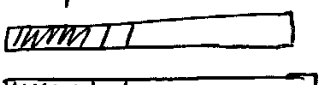
modified clock algo
first revolution: only look, don't ~~clear~~ ~~clear~~ clear ref. bits, but break if (0,0)
second revolution: look, and do clear ref. bits; if (0,0) or if (0,1) break

• additional-reference-bits/aging/reference bit history algorithm



bitshifts
periodically (every few clock ticks), do:
for each phys page, right shift by 1, entire arr val
copy PTE ref to high order bit
clear ref bit

on page fault, find min value in array; that's the victim

comparing A and B: A < B iff 

for all $j > i$, $A_j = B_j$ but $A_i == 0$ $B_i == 1$

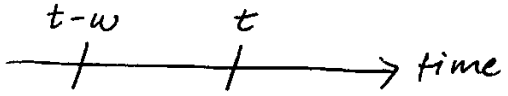
Global and Local Replacement

(Process-by-process alloc?)

global
Compete among all frames
Example FIFO: single FIFO list
performs better and more flexibly

local
compete among only own frames
Example FIFO: one FIFO list per process
more fair?

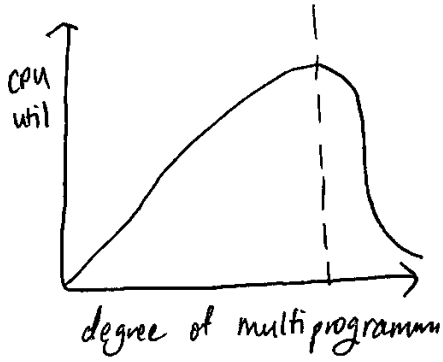
The working set: virtual page it is



$WS(T, w)$ = all pages sent over that time

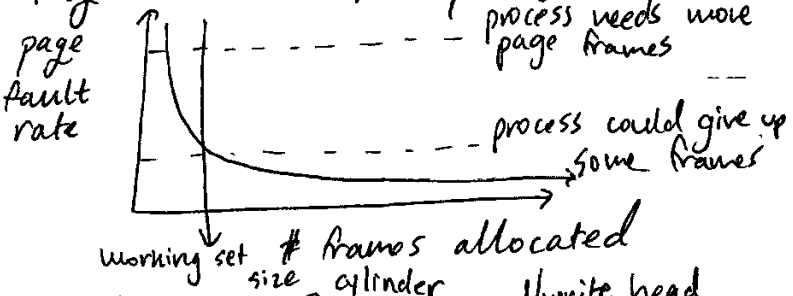
T: at time t; w: working at window size

Reference Bit History - any phys page while loading V bits are all 0

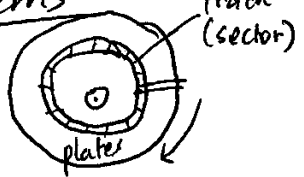


thrashing: spend more time on faults than useful executions
which process should be swapped out

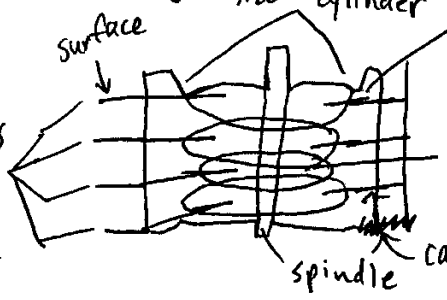
Page Fault Frequency Algos



File Systems



- Hard disk
surface is magnetically coated, hand reads magnetic signature
rotational speed not as easily controlled, read/write head can move in/out, must move seek to place on disk



"seek" in/out ↔ can move back and forth

TODO: seek to correct cylinder + select head (track) + rotational latency + bits of sector
to save time, save time needed for seeking in/out
for example

	old	common	fast
~~rpm~~			
rpm	3600	7200	10k
lrev	16⅔ ms	8⅓ ms	6 ms
ave. log latency	8⅓ ms	4⅓ ms	3 ms
seek avg	30 ms	9 ms	4 ms

Shrinithi Narayanan 19

"Geometry" (head #) x (cylinder #) x (sector/track #)

disk addr: (cylinder #, head #, sector #)

disk request scheduling (no longer just OSS job)

- automatic bad sector forwarding (eg more bits to farther cylinder to reduce seeking time)
- variable # sectors/track
- other secret geometries (disk companies competition)

new disk addressing: "Logical Sector Addressing" = sector #

- File Systems bits structure (on disk)
  - put "related" things "nearby" each other
  - structs are "permanent": bugs cause permanent defect
  - tradeoff between simplicity and performance

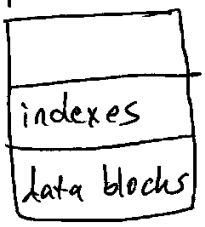
disc sector vs. file system blocks

- 512 bytes
- 4 KB
- 2^k bytes ← can lead to internal fragmentation

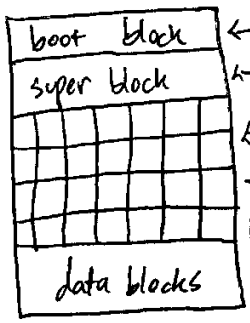
data structs on disk

- global: free space list
- ~~pfofi~~ profile: which blocks in this file

File system format



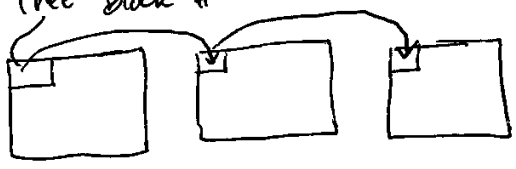
File System Format (classical Unix)



- boot block ← initial code
- super block ← global bookkeeping ← beginning of free list, # of total blocks
- ← "index nodes", ~~sort~~ sort of a page table, every file has an inode # corresponding to a block ← block to sector #
- other profile bookkeeping

free space management

1. in superblock; first free block #



normal linked list

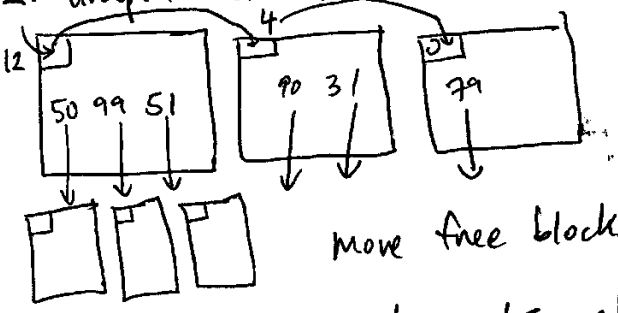
inefficient seek/lookup to disk for next free block

space taken up

→

Shrinith Narayana 20

2. "Grouped" Linked list



consume same amount of space
but fewer disk reads for more free block #'s

more free blocks once used, value is ∅ or 0

3. linked list w/ counts change each entry to (block #, count of contiguous free blocks)
but how to handle coalescing?

4. bitmap (1 bit/block)
0 = free, 1 = ~~not free~~ not free
4 kB block size = 4096 bytes = 32768 bits · 4096 = (large #) ← many blocks represented (1 bit/block)

per file block allocation "mode"

1. contiguous: in inode = (1st block #, # blocks, # bytes)
2. linked list: first block # in inode
   inefficient # of seeks to get next block
   ptr takes up space: powers of 2 broken (blocksize = 2^h - ptr size)

3. file allocation table (FAT): one global table = array of block #'s
   entries correspond to file #, first block # in inode 40, follow table ~~entries~~ entry #'s 40 → 5 → 100; alternate file eg 3 → 41
   similar to linked list, but table indexing much faster/efficient.
4. indexed allocation
   (a) in inode array of block #'s ← variable # of entries? (otherwise space wasted)
   fixed #: ~~waste~~ space wasted, unused
   (b) in inode: block # of index
   (c) in inode: block # of block of block #'s
5. combined scheme: handles files of all sizes w/ not much overhead

Naming: where is the name stored?
in the inode?
separate? stored where?
store directory entries same as file data ← dir has inode
dir entries in data blocks

structured (hierarchical) directories
ort hand. for pathnames, suppose lookup for
a/b/c/d/e/f/g/h
- hard to type, slow lookup for each directory down
- so, current directory at each process: in PCB: inode # of cur dir
looking up path name: starts with "/" (absolute pathname) → lookup start at constant root inode, otherwise use cur dir inode
to allow cd. and cd .., have dir entries for . and ..
multiple names for same file, references to same file from different locations. keep a reference count and free when ref count is 0. no easy way to check this problem.
no links to directories? link("old", "new") unlink("new")
mounting filesystems: how to name inodes unlink("old")
when one has multiple discs: each disk has its own inode #'s, starting at 0 or 1; mount(dev, dir, type, flags, opt)
dir goes to device
symbolic links symlink("old", "new") vs. hard links link("old", "new")
- way to distinguish original name
- way to detect when a cycle would happen when disconnected
- limit # of symlinks when traversing to prevent infinite loops
- when original link is deleted, other symlinks invalid

- no links to dirs
- can't tell which is old vs. new link

New Linux Filesystem - divide disk into cylinder groups

Shrin
Naray
22

- divide disk into cylinder groups
  in each: subset of data blocks, subset of inodes,
  list of free blocks, free inodes in cylinder group
  copy of superblock (all read only)

put non dir file inodes in same CG as dir inode
put new dir inode in new CG
put data blocks in same CG as inode (spread lead around)
  start new CG after all dir block #'s, and every 1 MB after
> limits # of seeks, backs up superblock
Multiple FS on same disk

---

how to track all these dynamically allocated inodes
NTFS (Windows NT filesystem)
  MFT (Master File Table) — First 16 MFT entries
    0 MFT itself
    3 root dirs
    6 bitmap of free space
    7 boot block
each MFT entry 1 KB
data block in each file sequence
of contigious chunks = "extent", each described by (VCN, LCN, len)
  VCN = virtual cluster # = block # within file
  LCN = logical cluster # = block # within whole FS
  len = size of block

Lab 3 Yalnix FS

just look at spec, order of operations important (wrong: might overwrite file after freeing inode w/ links pointing to it)
(could have unused space trapped/ alloced forever)

File Protection

In PCB, uid + gid
  user group

In inode, "mode" protection bits — RWX RWX RWX
  user group other
Ex. file mode 644 or 446
  dir mode 755 or 444
    "exec" opens dir
    ability to create files in dir

sticky bits: setuid setgid sticky bits
RWX RWX RWX
(restricted delete on dir) 1777 /tmp
  cannot rename or delete files in tmp

c. Protections

File	Dir
Read: read contents, view prot, owner, attrib	"list contents", view prot, owner attrib
Write: change contents, change prot, owner, attrib	change names in dir, change prot, owner, attr
Read/Exec: "read" and can exec (pass to kernel call)	"Read", list permissions, traverse
Modify: "write" and delete file	"Read+exec", delete dir
Full control: all of above	all above + below
List contents: X	list contents

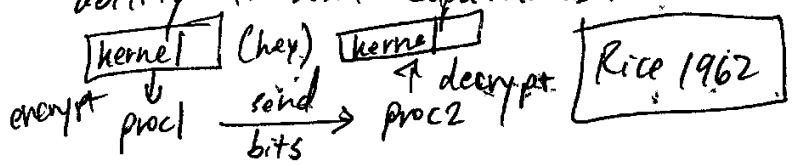
Protection domain = list of all "objects" and, foreach, list of "rights" on that object

could represent as access matrix: this matrix would be large + sparse

column = access control list (ACL) for that "object" (like window prot., unix a bits)
(what can others do to you)

row = capability list
(what can you do to other objects)

ability to send capabilities?



capability: secured name of some object and the rights to do various operations on that object (possession gives you the rights to do things on that object)

Program Security - Worms and Viruses

worm - independent program, attempts to spread self over network to other computers

virus - code in some program that, when ~~run~~ run, tries to spread itself by embedding in other programs

→

The Morris Internet Worm Nov 2 1998 from MIT Stim Nara

Three ways of spreading

1. Remote login ~~exec~~ - rlogin and rsh — try variations on username
   trusted computers, brute force typing passwords

2. Send mail DEBUG command (SMTP)
   prompts for password; the email addr = shell command
   default = "u" ↗ email body = std input
   often left unchanged
   runs as root admin access to remote comps] backdoor!

3. The "finger" Server
   Prevention gets → fgets. performs a buffer overflow
   check for buffer overflows strcpy → strncpy
   sprintf → snprintf

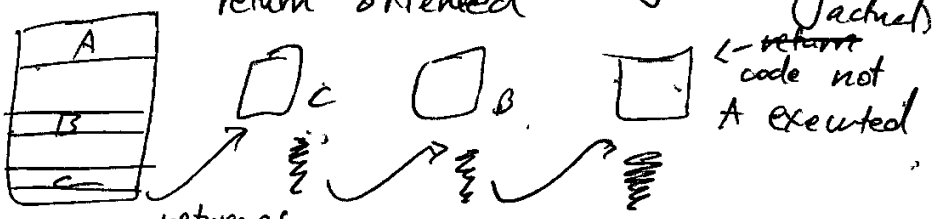
```
something C) {
  charbuf[512] buf;
  gets(buff);
  return;
```

Widely used, open ~~source~~ src or little snippets

Address Space Layout Randomization

load program @ random address, load libs @ random address
random space b/w stack frame, random malloc locations
compile to position independant code (PIC)

return to lib.c / return oriented programming



Stack Canaries

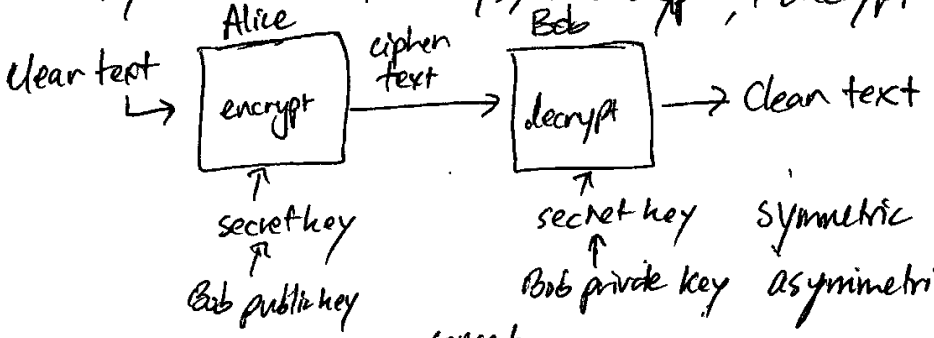
at program start, generate random #, store it "out of the way"; inside procedure: push that # onto stack, before return, check random # matches (random # relative to segment #, so can't know it's there)

...rity Services and Cryptography

Shrinithi Narayanan

confidentiality: protection against "eavesdropping"
authentication: confirm msg sent by who sender claims to be
integrity: confirm contents not modified

confidentiality – encryption
- symmetric – same key used for encrypt and decrypt (secret)
- asymmetric – two keys, 1 encrypt, 1 decrypt (public, private)



authentication
- symmetric: $E_{\text{secret A+B}}$ (data || Alice) || Alice — concat; encrypt w/ secret key; check Alice = Alice after decrypt
- asymmetric: $E_{\text{private A}}$ (data || Alice) || Alice — decrypt w/ public key (same can be done w/ integrity)

Secure Cryptographic One-Way Functions

bytes/bits → [hash func H(x)] → hashed value (fixed length)

Given $H(x) = y$, cannot find $z \neq x$ such that $H(z) = y$ as well

Example: MD5(x) = ~~128 bits~~ (not as secure) SHA-1(x) = 160 bits

authentication Alice

$M \rightarrow$ [hash H(x)] $\rightarrow H(M \,||\, \text{secret} \,||\, \text{Alice}) \,||\, M \,||\, \text{Alice}$ → [hash h(x)] → $H(M \,||\, \text{secret} \,||\, \text{Alice})$

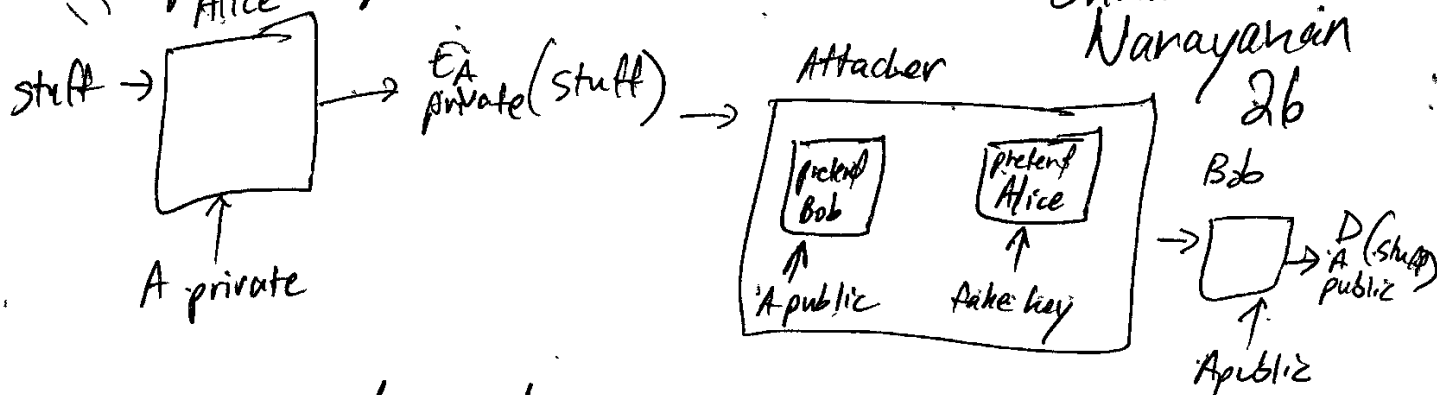
secret (for hash H(x)); secret (for hash h(x))

Benefits: reduces # of bits of msg

eg $E_{\text{private A's key}}$ (H(data || Alice)) || Alice || data

Attacking the Asymmetric Model (MITM)

Shrinithi Narayanan 26



the public key passed around can be hijacked

Public Key Certificates

certification authority = comodo, symantec, godaddy

- ~~certifac~~ certificate $X = (CA, Alice, Alice\ public\ key)$
  - CA: which cert authority
- use: $E_{CA\ private}(X) \| CA$ or $E_{CA\ private}(H(X)) \| X \| CA$

Passwords – prevent leaking, avoid expensive crypt/decrypt

- hashed passwords
  - store H(password)
  - check H(x) = H(password)
  - But same password means someone may login multiple places w/ same pwd
- add a "salt"
  - one random salt/user $H(password \| salt) \| salt$