e same hardware

Memory Multiplexing

Protection 防止对其它进程私有内存的访问

Controlled overlap 进程间共享内存

Translation: virtual memory → physical memory

1. give uniform view of memory to program
2. provide protection 3. control overlap

ultiplexing Memory Approaches

Uniprogramming only one program in the memory once
Not efficient (High overhead of OS in multiple program)
Not powerful (Cannot run multiple programs at a time)

Multiprogramming allow multiple programs using memory
n different blocks at the same time
More complex in OS (需要考虑 isolated memory 和 translation)

分配内存 (fixed-sized partition) 外部碎片 (dynamic partition)

Virtual Memory
- Base & Bound
- Segmentation
- Paging
- Paging + Segmentation

w Virtual Memory works?

CPU fetch instruction, 虚拟地址传入MMU进行地址转换
The memory returns the instruction
CPU decode instruction, use virtual address
With the help of MMU, the target memory is retrieved.

gmentation

ternal free gap between allocated chunks
ternal do not need all memory within allocated chunk

gmentation

ide memory into segmentation of varying length
ap virtual address in each segmentation to physical addr
: Easy to share, easy to protect, reduce fragmentation
: More complex, more overhead

aging

ide memory into frames of the same size
ap address by maintaining a page table
: Simple memory allocation, Easy to share
: Extral memory, need more time to access memory
ust be saved on Context Switch
Content of top-level segment registers
Pointer to top-level table
ache: a repository for copies that can be accessed
ore quickly than the origin

verage Access Time

t Rate x Hit Time) + (Miss Rate x Miss Time)

Temporal Locality

eep recently accessed data items closer to processor
patial Locality Move contiguous blocks to upper level

OS do on a Page Fault (Cache)

① Choose an old page to replace
② If old page modified, write content back to disk
③ Change its PTE and any cached TLB to be invalid
④ Load new page into memory from disk
⑤ Update page table entry, invalidate TLB for new entry
⑥ Continue thread from original faulting location.

Effective Access Time

EAT = Hit Time + Miss Rate × Miss Penalty

Miss Reason

Compulsory Miss Pages that never been paged into memory
Capacity Miss Not enough memory
Conflict Miss Miss happens but Cache is not full
Policy Miss Kicked out by the replacement policy

FIFO Throw out oldest page. Be fair - let every page live in memory for same amount of time. Bad - throw out heavily used pages instead of infrequently used.

MIN Replace page that will not be used for the longest time. Great, but cannot really know future.

Random Pick random page for every replacement. Pretty unpredictable - make it hard to make real-time guarantee.

LRU Replace page that has not been used for the longest time. Good Locality! Need to know immediately when each page used so that can change position in list

Clock Replace an old, not the oldest page.
Move slowly: not many page faults / find page quickly
Move quickly: lots of page faults / lots of reference bits

Add Memory doesn't always help page fault

Allocation

① **Equal allocation** Every process get same
② **Proportional allocation** $\frac{s_i}{\sum s_i} \times m$
③ **Priority allocation** according to the priority

Cache Tag n	Cache Index m	offset x
↓ Address number	↓ Block number	↓ Block size

Operational Parameter for I/O

① **Data granularity** Byte & Block
② **Access Pattern** Sequential & Random
③ **Transfer Mechanism** Programmed IO & DMA

How Processor talk to the device

① **CPU interacts with a Controller**
- Contain a set of registers that can be read/write
- Contain memory for request queue / bit-mapped image

② **Access Register**
- IO instruction

Programmed IO Each byte transferred via processor in/out or load/store
Pro: Simple hardware, easy to program
Con: Consumes processor cycles proportional to data size

Direct Memory Access Give controller access to memory bus, ask it to transfer data blocks to/from memory.
• CPU tell device driver to transfer disk data to buffer at address x • Device driver tells disk controller to transfer C bytes from disk to buffer at address x • Disk controller initiates DMA transfer • Disk controller send each byte to DMA controller • DMA controller transfer bytes to buffer x, increasing memory address and decreasing C until C=0 • When C=0, DMA interrupt CPU to signal transfer completion

IO device notify OS

The OS need to know when
- IO device has completed an operation
- IO device has encountered an error

IO interrupt: Device generated an interrupt whenever it need service.
• **Pro** handle unpredictable events well
• **Con** interrupt relatively high overhead

Polling: OS periodically check a device-specific status register
• **Pro** low overhead
• **Con** waste cycles on polling if unpredictable IO event

Device Drivers: Device-specific code in the kernel that interact directly with the device hardware
Top half accessed in call path from system call
Bottom half run as interrupt routine

Latency (Response Time) **Bandwidth (Throughput)**
S: start up cost B: transfer capacity n: length
Latency (n) = S + n/B **Bandwidth = n / Latency(n)**
half-power point: n = SB → Bandwidth = B/2

What determines Peak BW for IO
Bus Speed & Device Transfer Bandwidth

Three part of record on disk
Head, Track, Sector [identifier]

Read / Write data is a three-stage process:
① Seek time ② Rotational latency ③ Transfer Time

Disk Latency = Queueing Time + Controller Time + Seek Time + Rotational Time + Xfer Time
Avg (1/2)

SSD (No moving parts)
• Eliminates seek and rotational delay
• Very low power and lightweight • Limited write cycle

Contributing factor to latency
Software path, Hardware controller, IO device service time

FIFO SSTF SCAN C-SCAN LOOK C-LOOK

File System: Layer of OS that transforms block interface of disks into Files, Directories, etc.
① **Naming**: Interface to find files by name
② **Disk Management**: collecting disk blocks into files
③ **Protection**: Layer to keep data secure
④ **Reliability**: keep files despite crashes, failure, etc

Layout contiguous, linked, inode [allocation]

Contiguous Allocation
Pro Locate files easily, File deletion easily
Con external fragmentation

Linked Allocation
Pro reduce fragmentation, flexible
Con poor random access performance

FAT

Boot sector	FSINFO	FAT1	FAT2	Root Directory	File & dir

- Read root directory, retrieve first block number
- Read FAT determine the location of next block
- The process stops until FAT says the next block -1

FAT12 = 2^{12} **FAT16** = 2^{16} **FAT32** = 2^{28} clusters

Largest size of a FAT32 file = 4G - 1 bytes
FAT32 File system size = 2^{28} × block size

FAT series - directory entry
- stores the start cluster number
- stores the file size • store all file attributes

FAT series - reading a file
- Read the content from first cluster
- Look for the next cluster until reach the last

FAT series - writing a file
- Locate the last cluster
- start writing to the non-full cluster
- allocate the next cluster through FSINFO
- Update the FAT and FSINFO
- When writing finishes, update the file size

FAT series - deleting a file
- Deallocate all the blocks involved.
- Update FSINFO and FATs
- Change first byte of the directy entry to _ (0xE5)

File system
△ Transform blocks into Files and Directories
△ Optimize for access and usage pattern
△ Maximize sequential access, allow efficient random access

FAT scheme
△ Linked-list approach
△ Widely used: Cameras, USB drivers, SD cards

(Block size / Addr Length)^k × Block Size

Max file size != FS size

ext 2/3 - FS layout

Super block	GD PT	Reserve GDT	Block Bitmap	iNode Bitmap	iNode Table	Data Blocks

Why Ext having group?

Performance: Spatial locality

Reliability: superblock and GDT are replicated in each block group

Ext File deletion: Just an update of the "entry length" of previous entry

Hard link: a directory entry pointing to iNode of existing file, which can be accessed through two different pathname

Symbolic link: pointing to a new iNode whose target's pathname are stored using the space. [Map]

NTFS (tiny files data is in header)

Variable length extents

Everything is a sequence of <attribute:value> pair

Mix direct and indirect freely

Directories organized in B-tree structure by default

include

kernel manage all the physical address

system call configure kernel or build things

driver handle the interaction

shell render a simple command line

OS do

provide abstractions to apps • Manage resources

What is system call

It's a function call • exposed by the kernel

Abstract away most low-level details

Ext System call: printf, scanf, malloc, free, fopen, fclose

FS System call: mkdir, chown, chmod, open, getpid, write

Our Fundamental OS Concept

Thread • Single unique execution context, fully describe program state • PC, Register, Execution Flags, Stack

Address Space • Program execute in an address space that distinct from the memory space of the physical machine

Process • An instance of an executing program is a process consisting of an address space and 1 more thread of control

Dual mode operation/Protection • Only the system has the ability to access certain resources • The OS and the hardware protected from user programs and user programs are isolated from one another by controlling the translation from program virtual addresses to machine physical address

Virtual CPU need structure to hold PC, SP, Register

How switch from one CPU to the next?

Save PC, SP and register in current state block

3 types of Mode transfer: Unprogrammed

• System Call • interrupt • Trap or Exception

fork(): It creates the child process by cloning from the parent process, including all userspace data: PC[CPU reg], Program code [File & Memory], Memory, Opened Files [kernel] However, do not clone return value of fork(), PID, parent process, running time, File locks

exec(): The process is changing the code that is executing and never return to the origin code. Will replace: Program Code, Memory (local v, global v, dynamical memory), Reg value (PC)

wait(): suspend the calling process to waiting state & return when • one of its child (running → terminate) • signal is received • return immediately if it has no child.

fork() inside the kernel 1. copy kernel space 2. update kernel space (PID, Running time, pointer to parent/child) 3. copy userspace 4. return value ☆ Array of opened file → presene

exec() inside the kernel 1. clear local v & dynamical memory. reset global v based on new code, change constants to the new program code The kernel will also reset PC

exit() inside the kernel 1. clean up most of the allocated kernel space memory 2. clean up the exit process's userspace memory 3. notify the parent with SIGCHID → Zombie

wait() inside the kernel 1. register a signal handling routine 2. when SIGCHID comes, the corresponding signal handling routine is invoked (remove the signal, destroy the child process in the kernel space) 3. deregister the signal handling routine for the parent 4. return the PID of child

Exit() turn a process into a zombie when

1. The process calls exit() 2. The process return from main() ☆
3. The process terminates abnormally

When invoking a system call [kernel mode bit = 0 in kernel]

1. PC is pointing to the region (code) 2. CPU switches from userspace to kernel space, read the PID of process from mem. 3. it switches back to the userspace mem, continue running code.

Time

• User Time CPU time spent on code in userspace memory

• Sys Time CPU time spent on code in kernelspace memory

Context Switch: The scheduler decides to schedule another process in the ready queue. Then the schedule has to load the context of that process from the main mem to CPU [Expensive]: • Save & restore reg • Switch address space • Cache, buffer cache & TLB miss

[Save] - same process: TCB, reg, PC, SP

- different process: PCB, file-descriptor

CPU-bound Process: spend most of time in CPU, usertime > sys time [AI]

IO-bound Process: spend most of time in IO, sys time > user time ['ls']

Turnaround Time: Finish time - Arrive time

Waiting Time: Turnaround Time - task time ☆☆

RR Run: The responsiveness of the process is greater,

• PCB points to multiple TCB

• Switching thread within a block is simple

• Switching thread across blocks require changes to memory and IO address table

Multiprocessing ≡ Multiple CPU

Multiprogramming ≡ Multiple Job / Process

Multithreading ≡ Multiple thread per process

Multiprocessing A → / B →

Multiprogramming A → B →

Threading Model (user thread to kernel thread)

• One to One • Many to One • Many to Many

IPC

Shared Object ← FIFO / Pipe / Semaphore → system

Message Passing → user

Signal

Race Condition concurrently

• Happen whenever shared object + multiple process + execution sequences of the process

• The outcome of the computation depends on the

Mutual exclusion is a requirement

• If it could be achieved, the problem of race condition would be gone

A critical section

the code segment that access shared object can be designed for accessing more than one shared object.

Entry and Exit Implementation • Bounded Waiting • Progress

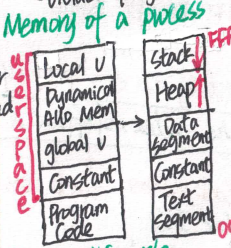
• Mutual Exclusion

Achieving Mutual Exclusion

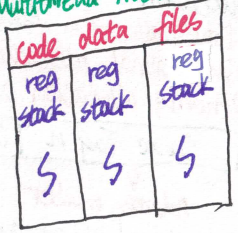
1. Spin-based lock 2. Spin Smarter 3. Semaphore

↳ violate progress ↳ suffer from priority inversion

Memory of a Process



Multithread Processes

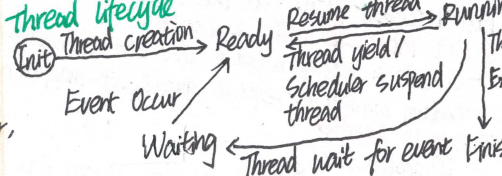


Process Lifecycle

Forked → Ready ⇌ Running → Zombie

Block / Waiting

Thread lifecycle



int turn; int interested[2] = {FALSE, FALSE};

void lock(int process){ int other; other = 1-process;

interested[process] = TRUE; turn = other; while(turn == other && interested[other]); }

void unlock(int process){ interested[process] = FALSE; }

→ A Low priority process L is inside the critical region, but a high priority process H gets the CPU and want to enter the critical region. But H cannot lock (since L not unlo

Producer-Consumer

SO: semaphore mutex = 1; avail = N; fill = 0;

ES: void producer{ int item; while(true){ item = produce wait(&avail); wait(&mutex); insert_item(item); post(&mutex); post(&fill); } }

ES: void consumer{ int item; while(true){ wait(&fill); wait(&mutex); item = remove_item(); post(&mutex); post(&avail); } }

Dining Philosopher (LEFT((i+N-1)%N) Right((i+1)%N

SO: int state[N]; semaphore mutex = 1, P[N] = 0

MF: void philosopher(int i){ think(); take(i); eat(); putt

SE: void take(int i){ wait(&mutex); state[i] = Hungry; captain(i); post(&mutex); wait(&P[i]); }

SE: void put(int i){ wait(&mutex); state[i] = Think; captain(LEFT); captain(RIGHT); post(&mutex); }

HF: void captain(int i){ if(state[i] = H && state[LL] != E state[R] != E){ state[i] = E; post(&P[i]); }

wait & post

void wait(semaphore *s){ disable_interrupt(); *s = *s- if(*s < 0){ enable_interrupt(); sleep(); disable_interr enable_interrupt(); }

void post(semaphore *s){ disable_interrupt(); *s = *s if(*s <= 0){ wake_up(); } enable_interrupt(); }

Four requirement for Deadlock

• Mutual Exclusion • Hold and Wait

• No preemption • Circular wait

Method of Handling Deadlock

• Allow system to enter deadlock and then recover • Ensure th system will never enter a deadlock • Ignore the deadlock

What to do when detect deadlock

• Terminat thread, force it to give up resources • Preempt resour without killing off process • Roll back action of deadlocked thread

Techique for Preventing Deadlock

• Infinite resource • No sharing of resource • Do not allow wait

• make all process request everything they need in the begining

• force all process to request resource in a particular order.

Thread State

• Shared: Heap, global v, Program code