Intro to C

Hello World

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

`#include`: preprocessor inserts `stdio.h` contents

`stdio.h`: contains `printf` declaration

`main`: program starts here

`void`: keyword for argument absence

`{ }`: basic block/scope delimiters

`printf`: prints to the terminal

`\n`: newline character

`return`: leave function, return value

Compiling

```
$ gcc hello.c -o hello
$ ./hello
Hello World!
```

Basic Data Types

`char c = 5; char c = 'a';`
one byte, usually for characters (1970: ASCII is fine)

`int i = 5; int i = 0xf; int i = 'a';`
usually 4 bytes, holds integers

`float f = 5; float f = 5.5;`
4 bytes floating point number

`double d = 5.19562`
8 bytes double precision floating point number

Basic Data Types — logic

`int i = 5 / 2; //i = 2`
integer logic, no rounding

`float f = 5.0f / 2; //f = 2.5f`
decimal logic for float and double

`char a = 'a'/ 2 //a = 97 / 2 = 48`
`char` interpreted as character by console

Basic Data Types — signed/unsigned

`signed int i = -5 //i = -5` (two's complement)

`unsigned int i = -5 //i = 4294967291`

Basic Data Types — short/long

`short int i = 1024 //-32768...32767`

`long int i = 1024 //-2147483648...2147483647`

Basic Data Types — more size stuff

`sizeof int; sizeof long int; //4; 4;` (x86 32-Bit)

use data types from `inttypes.h` to be sure about sizes:

```
#include <inttypes.h>
int8_t i; uint32_t j;
```

Basic Data Types — const/volatile

`const int c = 5;`
`i` is constant, changing it will raise compiler error

`volatile int i = 5;`
`i` is volatile, may be modified elsewhere (by different program in shared memory, important for CPU caches, register, assumptions thereof)

Variables — local vs. global

```
int m; // global variable

int myroutine(int j) {
  int i = 5 // local variable
  i = i+j;
  return i;
}
```

global variables (`int m`):
lifetime: while program runs
placed on pre-defined place in memory

basic block/function-local variables (`int i`):
lifetime: during invocation of routine
placed on stack or in registers

Variables — local vs. static

```
int myroutine(int j) {
  static int i = 5;
  i = i+j;
  return i;
}

k = myroutine(1); // k = 6
k = myroutine(1); // k = 7
```

static function-local variables:
saved like global variables
variable persistent across invocations
lifetime: like global variables

Printing

```
int i = 5; float f = 2.5;
printf("The numbers are i=%d, f=%f", i, f);
```

comprised of format string and arguments

may contain format identifiers (`%d`)

see also `man printf`

special characters: encoded via leading backslash:
`\n` newline
`\t` tab
`\'` single quote
`\"` double quote
`\0` `null`, end of string

Compound data types

`struct`ure: collection of named variables (different types)

`union`: *single* variable that can have multiple types

members accessed via `.` operator

```c
struct coordinate {
  int x;
  int y;
}

union longorfloat {
  long l;
  float f;
}

struct coordinate c;
c.x = 5;
c.y = 6;

union longorfloat lf;
lf.l = 5;
lf.f = 6.192;
```

Functions

encapsulate functionality (*reuse*)

code structuring (*reduce complexity*)

must be **declared** and **defined**

Declaration: states signature

Definition: states implementation (implicitly declares function)

```c
int sum(int a, int b); // declaration

int sum(int a, int b) { // definition
  return a+b;
}
```

Header files

header file for frequently used declarations

use `extern` to declare global variables defined elsewhere

use `static` to limit scope to current file (e.g. `static float pi` in `sum.c`: no `pi` in `main.c`)

```c
// mymath.h
int sum(int a, int b);
extern float pi;

// sum.c
#include "mymath.h"

float pi = 3.1415927;
int sum(int a, int b) {
  return a+b;
}

// main.c
#include <stdio.h>
#include "mymath.h"

void main() {
  printf("%d\n", sum(1,2));
  printf("%f\n", pi);
}
```

Data Segments and Variables

Stack: local variables

Heap: variables crated at runtime via `malloc()`/`free()`

Data Segment: static/global variables

Code: functions

Function overloading

no function overloading in C!

use arrays ore pointers

Pointers

```c
int a = 5;
int *p = &a // points to int, initalized to point to a
int *q = 32 // points to int at address 32
int b = a+1;
int c = *p; // dereference(p) = dereference(&a) = 5
int d = (*p)+2 // = 7
int *r = p+1; // pointing to next element p is pointing to
int e = *(p+2) // dereference (p+2) = d = 7
```

Pointers — linked list

linked-list implementation via next-pointer

```c
struct ll {
  int item;
  struct ll *next;
}

struct l first;
first.item = 123;

struct ll second;
second.item = 456;
first.next = &second;
```

Arrays

= fixed number of variables *continuously laid out in memory*

```c
int A[5]; // declare array (reserve memory space)
A[4] = 25; A[0] = 24; // assign 25 to last, 24 to first elem
char c[] = {'a',5,6,7,'B'} // init array, length implicit
c[64] = 'Z' // NO bounds checking at compile/run (may raise protection
      fault)

// declare pointer to array; address elements via pointer:
char *p = c;
*(p+1) = 'Z'; p[3] = 'B'; char b = *p; // = 'a'
```

Strings

= array of `char`s terminated by `NULL`:

```c
  char A[] = { 'T', 'e', 's', 't', '\0' };
  char A[] = "Test";
```

declaration via pointer:

```c
  const char *p = "Test";
```

common string functions (`string.h`):

length: `size_t strnlen(const char *s, size_t maxlen)`

compare:

```c
  int strncmp(const char *s1, const char *s2, size_t n);
```

copy: `int strncpy(char *dest, const char *src size_t n);`

tokenize: `char *strtok(char *str, const char *delim);`

(e.g. split line into words)

Arithmetic/bitwise operators

arithmetic operators:
`a+b`, `a++`, `++a`, `a+=b`, `a-b`, `a--`, `--a`, `a-=b`, `a*b`, `a*=b`, `a/b`, `a/=b`, `a%b`, `a%=b`

logical operators:
`a&b`, `a|b`, `a>>b`, `a<<b`, `a^b`, `~a`

difference pre-/post-increment:

```
int a = 5;
if(a++ == 5) printf("Yes"); // Yes
a = 5;
if(++a == 5) printf("Yes"); // nothing
```

operators in order of precedence:
`()`, `[]`, `->`, `.`
`!`, `++`, `--`, `+y`, `-y`, `*z`, `&=`, `(type)`, `sizeof`
`*`, `/`, `%`
`+`, `-`
`<<`, `>>`
`<`, `<=`, `>`, `>=`
`==`, `!=`
`&`
`^`
`|`
`&&`
`||`
`?`, `:`
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `~=`,=, «=, »=
`,`

Structures

brackets only needed for multiple statements

`if`/`else`, `for`, `while`, `do-while`, `switch`

may use `break`/`continue`

switch: need break statement, otherwise will fall through

```
if(a==b) printf("Equal") else printf("Different");
for(i=10; i>=10; i--) printf("%d", i+1);
int i=10; while(i--) printf("foo");
int i=0; do printf("bar"); while(i++ != 0);

char a = read();
switch(a) {
  case '1':
    handle_1();
    break;
  default:
    handle_other();
    break;
}
```

Type casting

explicit casting: precision loss possible
`int i = 5; float f = (float)i;`

implicit casting: if no precision is lost
`char c = 5; int i = c;`

pointer casting: changes address calculation
`int i = 5; char *p = (char *)&i; *(p+1)= 5;`

type hierarchy: „wider"/„shorter" types
`unsigned int` wider than `signed int`
operators cast parameters to widest type
Attention: assignment cast after operator cast

C Preprocessor

modifies *source code* before compilation

based on preprocessor *directives* (usually starting with `#`)

`#include <stdio.h>`, `#include "mystdio.h"`:
copies contents of file to current file

only works with strings in source file

completely ignores C semantics

Preprocessor — search paths

`#include <file>`: system include, searches in:
`/usr/local/include`
`libdir/gcc/[target]/[version]/include`
`/usr/[target]/include`
`/usr/include`
(target: arch-specific (e.g. i686-linux-gnu),
version: gcc version (e.g. 4.2.4))

`#include "file"`: local include, searches in:
directory containing current file
then paths specified by `-i <dir>`
then in system include paths

Preprocessor — definitions

defines introduce replacement strings (can have arguments, based on string replacement)

can help code structuring, often leading to source code cluttering

```
#define PI 3.14159265
#define TRUE (1)
#define max(a,b) ((a > b) ? (a) (b))
#define panic(str) do { printf(str); for (;;) } while(0);

#ifdef __unix__
# include <unistd.h>
#elif defined _WIN32
# include <windows.h>
#endif
```

Preprocessor — predefined macros

system-specific:
`__unix__`, `_WIN32`, `__STDC_VERSION__`

useful:
`__LINE__`, `__FILE__`, `__DATE__`

Libraries

= collection of functions contained in object files, glued together in dynamic/static library

ex.: Math header contains declarations, but not all definitions
⇝ need to link math library: `gcc math.c -o math -lm`

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

int main() {
  float f = 0.555f;
  printf("%f", sqrt(f*4));
  return 0;
}
```

Introduction to Operating Systems

What's an OS?

abstraction: provides abstraction for applications
- manages and hides hardware details
- uses low-level interfaces (not available to applications)
- multiplexes hardware to multiple programs (*visualization*)
- makes hardware use efficient for applications

protection:
- from processes using up all resources (*accounting*, *allocation*)
- from processes writing into other processes memory

resource managing:
- manages + multiplexes hardware resources
- decides between conflicting requests for resource use
- strives for efficient + fair resource use

control:
- controls program execution
- prevents errors and improper computer use

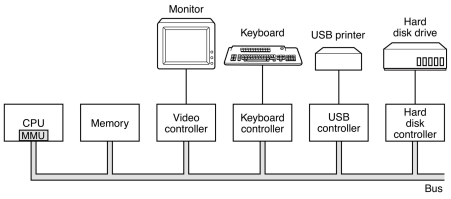
⇝ **no universally accepted definition**

Hardware Overview

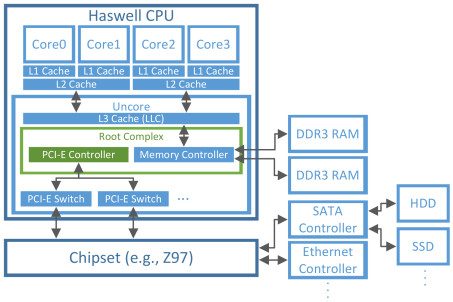
CPU(s)/devices/memory (conceptually) connected to common bus

CPU(s)/devices competing for memory cycles/bus

all entities run concurrently



today: multiple buses

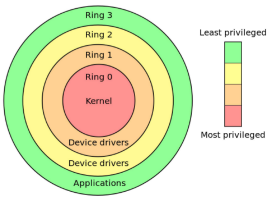


Central Processing Unit (CPU) — Operation

fetches instructions from memory, executes them (instruction format/-set depends on CPU)

CPU internal registers store (meta-)data during execution (general purpose registers, floating point registers, instruction pointer (IP), stack pointer (SP), program status word (PSW),...)

execution modes:
- **user mode** (x86: *Ring 3*/*CPL 3*):
  - only non-privileged instructions may be executed
  - cannot manage hardware ⟶ **protection**
- **kernel mode** (x86: *Ring 0*/*CPL 0*):
  - all instructions allowed
  - can manage hw with **privileged instructions**



Random Access Memory (RAM)

keeps currently executed instructions + data

today: CPUs have built-in *memory controller*

root complex connected directly via
- „wire" to caches
- pins to RAM
- pins to PCI-E switches

Caching

RAM delivers instructions/data slower than CPU can execute

memory references typically follow *locality principle*:
- **spatial locality**: future refs often near previous accesses (e.g. next byte in array)
- **temporal locality**: future refs often at previously accessed ref (e.g. loop counter)

caching helps mitigating this memory wall:
- copy used information temporarily from slower to faster storage
- check faster storage first before going down **memory hierarchy**
- if not, data is copied to cache and used from there

Access latency:
- register: ∼1 CPU cycle
- L1 cache (per core): ∼4 CPU cycles
- L2 cache (per core pair): ∼12 CPU cycles
- L3 cache/LLC (per uncore): ∼28 CPU cycles (∼25 GiB/s)
- DDR3-12800U RAM: ∼28 CPU cycles + ∼ 50ns (∼12 GiB/s)

Caching — Cache Organization

caches managed in hardware

divided into *cache lines* (usually 64 bytes each, unit at which data is exchanged between hierarchy levels)

often separation of data/instructions in faster caches (e.g. L1, see *Harvard architecture*)

cache hit: accessed data already in cache (e.g. L2 cache hit)

cache miss: accessed data has to be fetched from lower level

cache miss types:
- *compulsory miss*: first ref miss, data never been accessed
- *capacity miss*: cache not large enough for process working set
- *conflict miss*: cache has still space, but collisions due to placement strategy

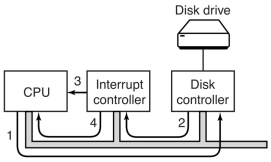
Interplay of CPU and Devices

I/O devices and CPU execute concurrently

Each device controller
- - is in charge of particular device
- - has local buffer

Workflow:

1. CPU issues commands, moves data to devices
2. Device controller informs APIC (*Advanced Programmable Interrupt Controller*) that operation is finished
3. APIC signals CPU
4. CPU receives device/interrupt number from APIC, executes handler



Device control

Devices controlled through their **device controller**, accepts commands from OS via **device driver**

devices controlled through device registers and device memory:
- *control* device by writing device registers
- *read* status of device by reading device registers
- *pass data* to device by reading/writing device memory

2 ways to access device registers/memory:

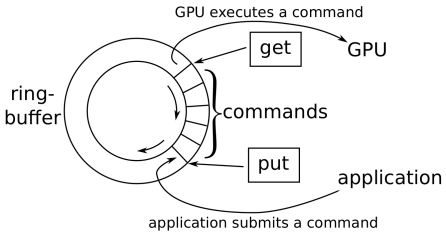
1. **port-mapped IO** (PMIO):
   use special CPU instructions to access port-mapped registers/memory
   e.g. x86 has different `in`/`out`-commands that transfer 1,2 or 4 bytes between CPU and device
2. **memory-mapped IO** (MMIO):
   use same address space for RAM and device memory
   some addresses map to RAM, others to different devices
   access device's memory region to access device registers/memory

some devices use hybrid approaches using both

Device control — nVidia general purpose GPU

memory-mapped ring-buffer and `put`/`get`-device

mapping can be exposed to application ⇝ application can submit commands in user-mode



Summary

The OS is an abstraction layer between applications and hardware (multiplexes hardware, hides hardware details, provides protection between processes/users)

The CPU provides a separation of User and Kernel mode (which are required for an OS to provide protection between applications)

CPU can execute commands faster than memory can deliver instructions/data – memory hierarchy mitigates this memory wall, needs to be carefully managed by OS to minimize slowdowns

device drivers control hardware devices through PMIO/MMIO

Devices can signal the CPU (and through the CPU notify the OS) through interrupts

OS Concepts

OS Invocation

OS Kernel does **not** always run in background!

Occasions invoking kernel, switching to kernel mode:

1. **System calls**: User-Mode processes require higher privileges
2. **Interrupts**: CPU-external device sends signal
3. **Exceptions**: CPU signals unexpected condition

System Calls — Motivation

Problem: protect processes from one another

Idea: Restrict processes by running them in user-mode

⇝ **Problem**: now processes cannot manage hardware,...
who can switch between processes?
who decides if process may open certain file?

⇝ **Idea**: OS provides **services** to apps
app calls system if service is needed (**syscall**)
OS checks if app is allowed to perform action
if app may perform action and hasn't exceeded quota,
OS performs action in behalf of app in kernel mode

System Calls — Examples

`fd = open(file, how,...)` – open file for read/write/both

documented e.g. in `man 2 write`

overview in `man 2 syscalls`

System Calls vs. APIs

syscalls: interface between apps and OS services, limited number of well-defined entry points to kernel

APIs: often used by programmers to make syscalls
e.g. `printf` library call uses `write` syscall

common APIs: Win32, POSIX, C API

System Calls — Implementation

trap instruction: single syscall interface (entry point) to kernel
switches CPU to kernel mode, enters kernel in same, predefined way for all syscalls
system call dispatches then acts as syscall multiplexer

syscalls identified by number passed to trap instruction
syscall table maps syscall numbers to kernel functions
dispatcher decides where to jump based on number and table
programs (e.g. `stdlib`) have syscall number compiled in!
⇝ never reuse old numbers in future kernel versions

Interrupts

devices use interrupts to signal predefined conditions to OS
reminder: device has „interrupt line" to CPU
e.g. device controller informs CPU that operation is finished

programmable interrupt controller manages interrupts
interrupts can be **masked**
masked interrupts: queued, delivered when interrupt unmasked
queue has finite length ⇝ interrupts can get lost

notable interrupt examples:

1. *timer-interrupt*: periodically interrupts processes, switches to kernel ⇝ can then switch to different processes for fairness

2. *network interface card* interrupts CPU when packet was received $\rightsquigarrow$ can deliver packet to process and free NIC buffer

when interrupted, CPU

1. looks up **interrupt vector** (= table pinned in memory, contains addresses of all service routines)
2. transfers control to respective **interrupt service routine** in OS that handles interrupt

interrupt service routine must first save interrupted process's state (instruction pointer, stack pointer, status word)

Exceptions

sometimes unusual condition makes it impossible for CPU to continue processing

$\rightsquigarrow$ **Exception** generated within CPU:

1. CPU interrupts program, gives kernel control
2. kernel determines reason for exception
3. if kernel can resolve problem $\rightsquigarrow$ does so, continues **faulting instruction**
4. kills process if not

Difference to Interrupts: interrupts can happen in any context, exceptions always occur asynchronous and in process context

OS Concepts — Physical Memory

up to early 60s:
- programs loaded and run directly in *physical memory*
- program too large $\rightarrow$ partitioned manually into *overlays*
- OS then swaps overlays between disk and memory
- different jobs could observe/modify each other

OS Concepts — Address Spaces

bad programs/people need to be isolated

Idea: give every job the illusion of having all memory to itself
every job has own *address space*, can't name addresses of others
jobs always and only use virtual addresses

Virtual Memory — indirect addressing

Today: every CPU has built-in **memory management unit** (*MMU*)

MMU translates virtual addresses to physical addresses at every store/load operation

$\rightsquigarrow$ address translation protects one program from another

Definitions:
Virtual address: address in process' address space
Physical address: address of real memory

Virtual Memory — memory protection

MMU allows kernel-only virtual addresses
- kernel typically part of all address spaces
- ensures that apps can't touch kernel memory

MMU can enforce *read-only* virtual addresses
- allows safe sharing of memory between apps

MMU can enforce execute disable
- makes code injection attacks harder

Virtual Memory — page faults

not all addresses need to be mapped at all times
- MMU issues *page fault* exception when accessed virtual address isn't mapped
- OS handles page faults by loading faulting addresses and then continuing the program
- $\rightsquigarrow$ memory can be **over-committed**: more memory than physically available can be allocated to application

page faults also issued by MMU on illegal memory accesses

OS Concepts — Processes

= program in execution („instance" of program)

each process is associated with a **process control block** (*PCB*)
contains information about allocated resources

each process is associated with a virtual **address space** (*AS*)
- all (virtual) memory locations a program can name
- starts at 0 and runs up to a maximum
- address 123 in AS1 generally $\neq$ address 123 in AS2
- indirect addressing $\rightsquigarrow$ different ASes to different programs
- $\rightsquigarrow$ protection between processes

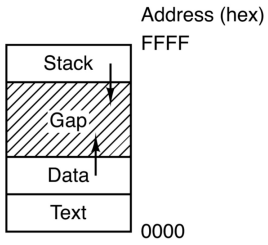
OS Concepts — address space layout

address spaces typically laid-out in different **sections**
- memory addresses between sections **illegal**
- illegal addresses $\rightsquigarrow$ page fault
- more specifically calls **segmentation fault**
- OS usually kills process causing segmentation fault

Stack: function history, local variables

Data: Constants, static/global variables, strings

Text: Program code



OS Concepts — Threads

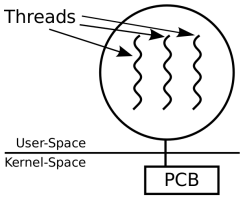
each progress: ≥ 1 threads (representing execution states)

IP stores currently executed instruction (address in `text` section)

SP register stores address of stack top
(> 1 threads $\rightarrow$ multiple stacks!)

PSW contains flags about execution history
(e.g. last calculation was 0 $\rightarrow$ used in following jump instruction)

more general purpose registers, floating point registers,...



OS Concepts — Policies vs. Mechanisms

separation useful when designing OS

Mechanism: implementation of what is done
(e.g. commands to put a HDD into standby mode)

Policy: rules which decide when what is done and how much
(e.g. how often, how many resources are used,...)

$\rightarrow$ *mechanisms can be reused even when policy changes*

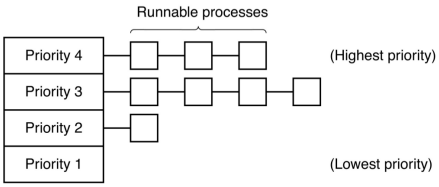
OS Concepts — Scheduling

multiple processes/threads available ⇝ OS needs to switch between them (for multitasking)

scheduler decides which job to run next (policy)

dispatcher performs task-switching (mechanism)

schedulers try to
- provide fairness
- while meeting goals
- and adhering to priorities



OS Concepts — Files

OS hides peculiarities of disks,...

programmer uses device-independent *files*/*directories* for persistent storage

Files: associate *file name* and *offset* with bytes

Directories: associate *directory names* with directory names or file names

File System: ordered block collection
- main task: translate (dir name + file name + offset) to block
- programmer uses file system operations to operate on files (`open`, `read`, `seek`)

processes can communicate directly through special *named pipe* file (used with same operations as any other file)

OS Concepts — Directory Tree

directories form *directory tree*/*file hierarchy*
→ structure data

root directory: topmost directory in tree

files specified by providing *path name* to file

OS Concepts — Mounting

Unix: common to orchestrate multiple file systems in single file hierarchy

file systems can be *mounted* on directory

Win: manage multiple directory hierarchies with drive letters (e.g. `C:\Users`)

OS Concepts — Storage Management

OS provides uniform view of information storage to file systems
- *drivers* hide specific hardware devices
  → hides device peculiarities
- general interface abstracts physical properties to logical units
  → block

OS increases I/O performance:
- **Buffering**: Store data temporarily while transferred
- **Caching**: Store data parts in faster storage
- **Spooling**: Overlap one job's output with other job's input

Summary

OS provides abstractions for and protection between applications

kernel does not always run — certain events invoke kernel
- — *syscall*: process asks kernel for service
- — *interrupt*: device sends signal that OS has to handle
- — *exception*: CPU encounters unusual situation

processes encapsulate resources needed to run program in OS
- — *threads*: represent different execution states of process
- — *address space*: all memory process can name
- — *resources*: allocated resources, e.g., open files

scheduler decides which process to run next when multi-tasking

virtual memory implements address spaces, provides protection between processes

Processes

The Process Abstraction

computers do „several things at the same time“ (just looks this way though quick process switching (**Multiprogramming**))

⇝ **process** abstraction models this concurrency:
- container contains information about program execution
- conceptually, every progress has own „virtual CPU“
- execution context is changed on process switch
- dispatcher switches context when switching processes
- **context switch**: dispatcher saves current registers/memory mappings, restores those of next process

Process-Cooking Analogy

Program/Process like Recipe/Cooking

Recipe: lists ingredients, gives algorithm what to do when
⇝ program describes memory layout/CPU instructions

Cooking: activity of using the recipe
⇝ process is activity of executing a program

multiple similar recipes for same dish
⇝ multiple programs may solve same problem

recipe can be cooked in different kitchens at the same time
⇝ program can be run on different CPUs at the same time (as different processes)

multiple people can cook one recipe
⇝ one process can have several worker threads

Concurrency vs. Parallelism

OS uses currency + parallelism to implement multiprogramming

1. **Concurrency**: multiple processes, one CPU
   ⇝ not at the same time
2. **Parallelism**: multiple processes, multiple CPU
   ⇝ at the same time

Virtual Memory Abstraction — Address Spaces

every process has own *virtual addresses* (vaddr)

MMU relocates each load/store to *physical memory* (pmem)

processes never see physical memory, can't access it directly

\+ MMU can enforce protection (mappings in kernel mode)

\+ programs can see more memory than available
  80:20 rule: 80% of process memory idle, 20% active
  can keep working set in RAM, rest on disk

\- need special MMU hardware

Address Space (Process View)

code/data/state need to be organized within process
  ⇝ **address space layout**

Data types:

1. *fixed size* data items
2. data naturally *freed in reverse allocation order*
3. data *allocated/freed „randomly“*

compiler/architecture determine how large int is and what instructions are used in text section (`code`)

Loader determines based on exe file how executed program is placed in memory

Segments — Fixed-Size Data + Code

some data in programs never changes or will be written but never grows/shrinks
  ⇝ memory can be statically allocated on process creation

BSS segment (*block started by symbol*):
  - statically allocated variables/non-initialized variables
  - executable file typically contains starting address + size of BSS
  - entire segment initially 0

Data segment:
  - fixed-size, initialized data elements (e.g. global variables)

read-only data segment:
  - constant numbers, strings

All three sometimes summarized as one segment

compiler and OS decide ultimately where to place which data/how many segments exist

Segments — Stack

some data naturally freed in reverse allocation order
  ⇝ very easy memory management (stack grows upwards)

fixed segment starting point

store top of latest allocation in **stack pointer** (SP)
  (initialized to starting point)

allocate `a` byte data structure: `SP += a; return(SP - a)`

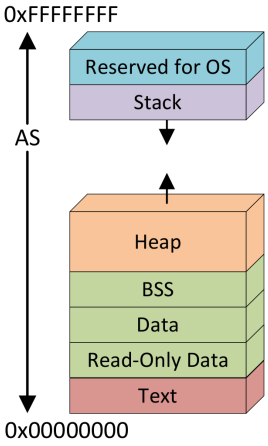
free `a` byte data structure: `SP -= a`

Segments — Heap (Dynamic Memory Allocation)

some data „randomly“ allocated/freed

two-tier memory allocation:

1. allocate large memory chunk (**heap segment**) from OS
   - base address + **break pointer** (BRK)
   - process can get more/give back memory from/to OS
2. dynamically partition chunk into smaller allocations
   - `malloc`/`free` can be used in random oder
   - purely user-space, no need to contact kernel



Summary

recipe vs. cooking is like program vs. process

processes = resource container for OS

process feels alone: has own CPU + memory

OS implements multiprogramming through rapid process switching

Process API

Execution Model — Assembler (Simplified)

OS interacts directly with compiled programs
  - switch between processes/threads ⇝ **save**/**restore** state
  - deal with/pass on **signals**/**exceptions**
  - receive **requests** from applications

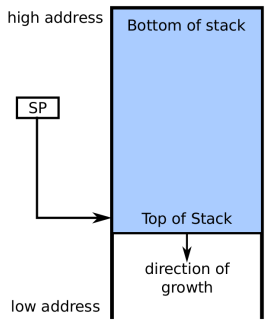
Instructions:
  - `mov`: Copy referenced data from second operand to first operand
  - `add`/`sub`/`mul`/`div`: Add,…from second operand to first operand
  - `inc`/`dec`: increment/decrement register/memory location
  - `shl`/`shr`: shift first operand left/right by amount given by second operand
  - `and`/`or`/`xor`: calculate bitwise and,…of two operands storing result in first
  - `not`: bitwise negate operand

Execution Model — Stack (x86)

stack pointer (SP): holds address of stack top (growing downwards)

stack frames: larger stack chunks

base pointer (BP): used to organize stack frames



Execution Model — jump/branch/call commands (x86)

`jmp`: continue execution at operand address

`j$condition`: jump depending on PSW content
- true ⇝ jump
- false ⇝ continue
- examples: `je` (jump equal), `jz` (jump zero)

`call`: push function to stack and jump to it

`return`: return from function (jump to return address)

Execution Model — Application Binary Interface (ABI)

standardizes binary interface between programs, modules, OS:
- executable/object file layout
- calling conventions
- alignment rules

calling conventions: standardize exact way function calls are implemented
⇝ interoperability between compilers

Execution Model — calling conventions (x86)

function call (caller):

1. save local scope state
2. set up parameters where function can find them
3. transfer control flow

function call (called function):

1. set up new local scope (local variables)
2. perform duty
3. put return value where caller can find it
4. jump back to caller (IP)

Passing parameters to the system

parameters are passed through **system calls**

call number + specific parameters must be passed

parameters can be transferred through
- **CPU registers** (~6)
- **Main Memory** (heap/stack – more parameters, data types)

ABI specifies how to pass parameters

return code needs to be returned to application
- **negative**: error code
- **positive + 0**: success
- usually returned via A+D registers

System call handler

implements the actual service called through a syscall:

1. saves tainted registers
2. reads passed parameters
3. sanitizes/checks parameters
4. checks if caller has enough permissions to perform the requested action
5. performs requested action in behalf of the caller
6. returns to caller with success/error code

Process API — creation

process creation events:

1. system initialization
2. process creation syscall
3. user requests process creation
4. batch job-initiation

events map to two mechanisms:

1. Kernel spawns initial user space process on boot (Linux: `init`)
2. User space processes can spawn other processes (within their quota)

Process API — creation (POSIX)

`PID`: identifies process

`pid = fork()`: duplicates current process:
- returns 0 to new child
- returns new `PID` to parent
- ⇝ child and parent independent after `fork`

`exec(name)`: replaces own memory based on executable file
- `name` specifies binary executable file

`exit(status)`: terminates process, returns `status`

`pid = waitpid(pid, &status)`: wait for child termination
- `pid`: process to wait for
- `status`: points to data structure that returns information about the process (e.g., exit status)
- passed `pid` is returned on success, `-1` otherwise

process tree: processes create child processes, which create child processes, …
- parent and child execute concurrently
- parent waits for child to terminate (collecting the exit state)

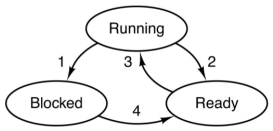
Daemons

= program designed to run in the background

detached from parent process after creation, reattached to process tree root (`init`)

Process States

blocking: process does nothing but wait
- usually happens on syscalls (OS doesn't run process until event happens)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process Termination

different termination events:

1. normal exit (voluntary)
   - `return 0` at end of `main`
   - `exit(0)`
2. error exit (voluntary)
   - `return x` ($x \neq 0$) at end of `main`
   - `exit(x)` ($x \neq 0$)
   - `abort()`
3. fatal error (involuntary)
   - OS kills process after exception
   - process exceeds allowed resources
4. killed by another process (involuntary)
   - another process sends kill signal (only as parent process or administrator)

Exit Status

voluntary exit: process returns exit status (integer)

resources not completely freed after process terminates
⇝ **Zombie** or **process stub** (contains exit status until collected via `waitpid`)

Orphans: Processes without parents
- usually adopted by `init`
- some systems kill all children when parent is killed

exit status on involuntary exit:
- Bits 0–6: signal number that killed process (0 on normal exit)
- Bit 7: set if process was killed by signal
- Bits 8–15: 0 if killed by signal (exit status on normal exit)

Threads

Processes vs. Threads

Traditional OS: each process has
- own address space
- own set of allocated resources
- *one* thread of execution (= one execution state)

Modern OS: processes + threads of execution handled more flexibly
- *processes* provide abstraction of address space and resources
- *threads* provide abstraction of execution states of that address space

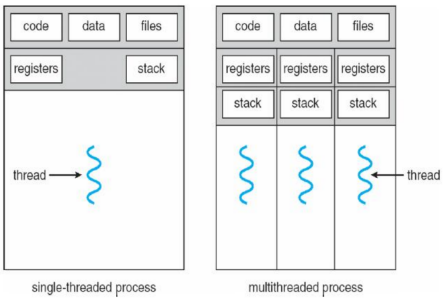
Exceptions:
- sometimes different threads have different address spaces
- *Linux*: threads = regular processes with shared resources and AS regions

Threads — why?

many programs do multiple things at once (e.g. web server)
⇝ writing program as many sequential threads may be easier than with blocking operations

Processes: rarely share data (if, then explicitly)

Threads: closely related, share data



Threads — POSIX

`PThread`: base object with
- *identifier* (thread ID, TID)
- *register set* (including IP and SP)
- *stack area* to hold execution state

`Pthread_create`: create new thread
- Pass: *pointer* to `pthread_t` (will hold TID after successful call)
- Pass: *attributes*, *start function*, *arguments*
- Returns: 0 on success, error value else

`Pthread_exit`: terminate calling thread
- Pass: exit code (casted to void pointer)
- Free's resources (e.g. stack)

`Pthread_join`: wait for specified thread to exit
- Pass: `ptread_t` to wait for (or -1 for any thread)
- Pass: pointer to pointer for exit code
- Returns: 0 on success, error value else

`Pthread_yield`: release CPU to let another thread run

Threads — Problems

Processes vs. Threads:
- *Processes*: only share resources explicitly
- *Threads*: more shared state ⟶ more can go wrong

Challenges: programmer needs to take care of
- *activities*: dividing, ordering, balancing
- *data*: dividing
- *shared data*: access synchronizing

PCB vs. TCP

PCB (*process control block*): information needed to implement processes
- always known to OS

TCB (*thread control block*): per thread data
- OS knowledge depends on *thread model*

PCB	TCB
Address space	Instruction pointer
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	

Thread models

Kernel Thread: known to OS kernel

User Thread: known to process

N:1-Model: kernel only knows one of possibly multiple threads
- N:1 user threads = *user level threads* (ULT)

1:1-Model: each user thread maps to one kernel thread
- 1:1 user threads = *kernel level threads* (KLT)

M:N-Model (hybrid model): flexible mapping of user threads to less kernel threads

Thread models — N:1

Kernel only manages process ⟶ multiple threads unknown to kernel

Threads managed in user-space library (e.g. GNU Portable Threads)

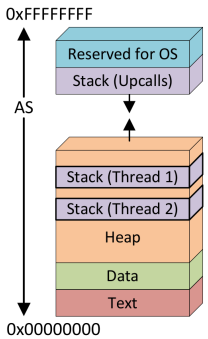
Pro:
+ faster thread management operations (up to 100 times)
+ flexible scheduling policy
+ few system resources
+ usable even if OS doesn't support threads

Con:
- no parallel execution
- whole process blocks if one user thread blocks
- reimplementing OS parts (e.g. scheduler)

Stack:
- main stack known to OS used by thread library
- own execution state (= stack) dynamically allocated by user thread library for each thread
- possibly own stack for each exception handler

Heap:
- concurrent heap use possible
- *Attention*: not all heaps are reentrant

Data: divided into BSS, data and read-only data here as well



Thread models — 1:1

kernel knows + manages every thread

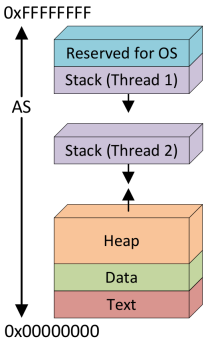
Pros:
+ real parallelism possible
+ threads block individually

Cons:
- OS manages every thread in system (TCB, stacks,…)
- Syscalls needed for thread management
- scheduling fixed in OS

Stack:
- own execution state (= stack) for every thread
- possibly own stack for (each) exception handler

Heap:
- parallel heap use possible
- *Attention*: not all heaps are thread-safe
- if thread-safe: not all heap implementations perform well with many threads

Data: divided into BSS, data and read-only data here as well



Thread models — M:N

Principle: M ULTs are maps to (at most) N KLT
- *Goal*: pros of ULT and KLT — non-blocking with quick management
- create sufficient number of KLTs and flexibly allocate ULTs to them
- *Idea*: if ULT blocks ULTs can be switched in userspace

Pros:
+ flexible scheduling policy
+ efficient execution

Cons:
- hard to debug
- hard to implement (e.g. blocking, number of KLTs,…)

Implementation — Up-calls:
- kernel notices that thread will block → sends signal to process
- up-call notifies process of thread id and event that happened
- exception handler of process schedules a different process thread
- kernel later informs process that blocking event finished via other up-call

Summary

programs often do closely related things at once
— mapped to thread abstraction: multiple threads of execution operate in same process

differentiation between process information (PCB) and thread information (TCB)

thread models:
— $N:1$: threads fully managed in user-space
— $1:1$: threads fully managed by kernel
— $M:N$: threads are flexibly managed either in user-space or kernel

multi-threaded programs operate on same data concurrently or even in parallel:
— *synchronization*: accessing such data must be synchronized
→ makes writing such programs challenging

Scheduling

Motivation

K jobs ready to run, $K > N \geq 1$ CPUs available

Scheduling Problem:
- Which jobs should kernel assign to which CPUs?
- When should it make decision?

Dispatcher

Dispatcher: performs actual process switch
- mechanism
- save/restore process context
- switch to user mode

Scheduler: selects next process to run based on *policy*

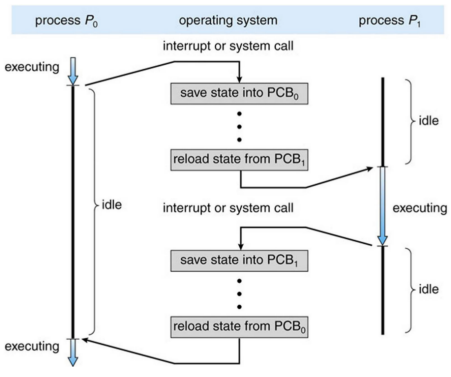
Voluntary Yielding vs. Preemption

kernel responsible for CPU switch

kernel doesn't always run → can only dispatch different process when invoked

cooperative multitasking: running process performs *yield* syscall → kernel switches process

preemptive scheduling:
- kernel invoked in certain time intervals
- kernel makes scheduling decisions after every time-slice



Scheduling — Process States

new: process was created but did not run yet

running: instructions are currently being executed

waiting: process is waiting for some event

ready: process is waiting to by assigned a processor

terminated: process has finished execution

Scheduling — long-term vs. short-term

Short-term scheduler (CPU Scheduler, focused on in this lecture):
- selects process to run next, allocates CPU
- invoked frequently (ms) $\rightsquigarrow$ must be fast

Long-term scheduler (job scheduler):
- selects process to be brought into ready queue
- invoked very infrequently (s, m) $\rightsquigarrow$ can be slow
- controls degree of *multiprogramming*

Scheduling queues

job queue: set of all processes in system

ready queue: process in main memory, ready or waiting

device queue: processes waiting for I/O device

Scheduling Policies — Categories

batch scheduling:
- still widespread in business (payroll, inventory,…)
- no users waiting for quick response
- non-preemptive algorithms acceptable $\rightarrow$ less switches $\rightarrow$ less overhead

interactive scheduling:
- need to optimize for response time
- preemption essential to keep processes from hogging CPU

real-time scheduling:
- guarantee job completion within time constraints
- need to be able to plan when which process runs + how long
- preemption not always needed

Scheduling Policies — Goals

General:
- *fairness*: give each process fair share of CPU
- *balance*: keep all parts of system busy

batch scheduling:
- *throughput*: number of processes that complete per time unit
- *turnaround time*: time from job submission to job completion
- *CPU utilization*: keep CPU as busy as possible

interactive scheduling:
- *waiting time*: reduce time a process waits in waiting queue
- *response time*: time from request to first response

real-time scheduling:
- *meeting deadlines*: finishing jobs in time
- *predictability*: minimize jitter

Scheduling Policies — first come first served

intuitively clear

Example: 3 processes arrive at time 0 in the order P_1, P_2, P_3

Process	Burst time	Turnaround time
P_1	24	24
P_2	3	27
P_3	3	30

$\rightsquigarrow$ average turnaround time 27 $\rightarrow$ *can we do better?*

Conclusion: if processes would arrive in order P_2, P_3, P_1, average turnaround time would be 13
$\rightsquigarrow$ *good scheduling can reduce turnaround time*

Scheduling Policies — shortest job first

Benefits: optimal average turnaround/waiting/response time

Challenge: cannot know job lengths in advance

Solution: predict length of next CPU burst for each process
$\rightsquigarrow$ schedule process with shortest burst next

Burst Estimation: *exponential averaging*
- $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
(t_n: actual length of n-th CPU burst, τ_{n+1}: predicted length of next CPU burst, $0 \leq \alpha \leq 1$)

Process Behavior — CPU bursts

CPU bursts exists because processes wait for I/O

CPU-bound processes: spends more time doing computations
$\rightsquigarrow$ few very long CPU bursts

I/O-bound processes: spends more time doing I/O
$\rightsquigarrow$ many short CPU bursts

Scheduling Policies — preemptive shortest-job-first

SJF optimizes waiting/response time
$\rightsquigarrow$ what about throughput?

Problem: CPU-bound jobs hold CPU until exit or I/O $\rightarrow$ *poor I/O utilization*

Idea: SJF, but preempt periodically to make new scheduling decision
- each time slice: schedule job with shortest remaining time next
- alternatively: schedule job with shortest next CPU burst

Scheduling Policies — round robin

Problem: batch schedulers suffer from starvation and don't provide fairness

Idea: each process runs for small CPU time unit
- *time quantum*/*time slice* length: usually 10-100ms
- preempt processes that have not blocked by end of time slice
- append current thread to end of run queue, run next thread

Caution: time slice length needs to balance interactivity and overhead!
$\rightarrow$ if time slice length in the area of dispatch time, 50% of CPU time wasted for process switching

Scheduling Policies — virtual round robin

Problem: RR is unfair for I/O-bound jobs: they block before using up time quantum

Idea: put jobs that didn't use up their quantum in additional queue
- store share of unused time-slice
- give those jobs additional queue priority
- put them back into normal queue afterwards

Scheduling Policies — (strict) priority scheduling

Problem: not all jobs are equally important
$\rightsquigarrow$ different priorities (e.g., 4)

Solution: associate priority number with each process
- RR for each priority
- *aging*: old low priority processes get executed before new higher priority processes

Scheduling Policies — multi-level feedback queue

Problem: context switching expensive
$\rightsquigarrow$ trade-off between interactivity and overhead?

Goals:
- higher priority for I/O jobs (usually don't use up quantum)
- low priority for CPU jobs (rather run them longer)

Idea: different queues with different priorities and time slice lengths
- schedule queues with (static) priority scheduling
- double time slice length in each next-lower priority
- process to higher priority when they don't use up quantum repetitively
- process to lower priority when they use up quantum repetitively

Scheduling Principles — priority donation

Problem: Process B (higher priority) waits for process A (lower priority)
$\rightsquigarrow$ B has now effectively lower priority

Solution: *priority donation*
- give A priority of B as long as B waits for A
- if C, D, E wait for B $\rightarrow$ A gets highest priority of B, C, D, E

Scheduling Policies — lottery scheduling

issue number of lottery tickets to processes (amount depending on priority)

amount of tickets controls average proportion of CPU for each process

Scheduling: scheduler draws random number N, process with N-th ticket is executed

processes can transfer tickets to other processes if they wait for them

Summary

phases: processes have phases of communication and waiting for I/O
$\rightarrow$ appropriate switching between processes increases computing system utilization

goal-based: scheduler decides what appropriate means based on goals
— *long-term scheduler*: degree of multiprogramming
— *short-term scheduler*: which process to run next

dispatching: only happens when OS is invoked
— *cooperative scheduling*: currently running thread yields (syscall)
— *preemptive scheduling*: OS is called periodically to switch threads

Inter Process Communication

Overview

Reasons for cooperating processes:
- *information sharing*: share file/data-structure in memory
- *computation speed-up*: break large tasks in subtasks $\rightsquigarrow$ parallel execution
- *modularity*: divide system into collaborating modules with clean interfaces

IPC: allows data exchange
- *message passing*: explicitly send/receive information using syscalls
- *shared memory*: physical memory region used by multiple processes/threads

IPC — message passing

= mechanism for processes to communicate and synchronize

message passing facilities generally provide `send` and `receive`

Implementations:
- hardware bus
- shared memory
- kernel memory
- network interface card (NIC)

Direct messages: processes explicitly named when exchanging messages

Indirect messages: sending to/receiving from *mailboxes*
- first communicating process creates mailbox, last destroys
- processes can only communicate through shared mailbox

Indirect messages – synchronization

Blocking (synchronous):
- *blocking send*: sender blocks until message is received
- *blocking receive*: receiver blocks until message is available

Non-blocking (asynchronous):
- *non-blocking send*: sender sends message, then continues
- *non-blocking receive*: receiver receives valid message or `null`

Messaging — Buffering

messages are *queued* using different capacities while being in-flight

zero capacity: no queuing
- *rendezvous*: sender must wait for receiver
- message is transferred as soon as receiver becomes available $\rightarrow$ no latency/jitter

bounded capacity: finite number + length of messages
- sender can send before receiver waits for messages
- sender must wait if link is full

unbounded capacity:
- sender never waits
- memory may overflow $\rightarrow$ potentially large latency/jitter between `send` and `receive`

Messaging — POSIX message queues

create or open existing message queue:
```
mqd_t mq_open (const char *name, int oflag);
```
- `name` ist path in file system
- access permission controlled through file system access permission

send message to message queue:
```
int mq_send (mqd_t md, const char *msg, size_t len, unsigned priority);
```

receive message with highest priority in message queue:
```
int mq_receive (mqd_t md, char *msg, size_t len, unsigned *priority);
```

register callback handler on message queue (to avoid polling):
```
int mq_notify (mqd_t md, const struct sigevent *sevp);
```

remove message queue:
```
int mq_unlink (const char *name);
```

Shared Memory

Principle: communicate through region of shared memory
- every write to shared region is visible to all other processes
- hardware guarantees that always most recent write is read

Implementation: message passing via shared memory is application-specific

Problems: using shared memory in a safe way is tricky
- *cache coherency protocol*: makes usage with many processes/CPUs hard
- *race conditions*: makes usage with multiple writers hard

Shared Memory — POSIX shared memory

create or open existing POSIX shared memory object:
```
int shm_open (const char *name, int oflag, mod_t mode);
```

set size of shared memory region:
```
ftruncate (smd, size_t len);
```

map shared memory object to address space:
```
void* mmap (void* addr, size_t len, [...], smd, [...]);
```

unmap shared memory object from address space:
```
int munmap (void* addr, size_t len);
```

destroy shared memory object:
```
int shm_unlink (const char *name);
```

Shared Memory — sequential memory consistency

= *the result of execution as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program.*

Model:
- all memory operations occur one at a time in *program order*
- ensures write atomicity

Reality: compiler and CPU re-order instructions to *execution order*
$\rightarrow$ without SC many processes on many CPU behave worse than preemptive threads on 1 CPU

Shared Memory — memory consistency model

Problem:
- CPUs generally not sequentially consistent
- compilers do not generate code in program order

Synchronization — race conditions

Assume: sequential memory consistency ⟶ no atomic memory transactions!

Critical Sections: protect instructions inside critical section from concurrent execution

Critical Sections — desired properties

mutual exclusion: at most one thread can be in the CS at any time

progress: no thread running outside of CS may block another thread from getting in

bounded waiting: once a thread starts trying to enter CS, there is a bound on number of times other threads get in

Critical Sections — disabling interrupts

kernel only switches on interrupts (usually on *timer interrupt*)
⟶ have per-thread *do not interrupt* (DNI)-bit

single-core system:
- enter CS: set DNI bit
- leave CS: clear DNI bit

Advantages:
- easy + convenient in kernel

Disadvantages:
- *only works on single-core systems*: disabling interrupts on one CPU doesn't affect other CPUs
- *only feasible in kernel*: don't want to give user power to turn off interrupts!

Critical Sections — lock variables

define global `lock` variable
- only enter CS if `lock` is 0, set to 1 on enter
- wait for lock to become 0 otherwise (*busy waiting*)

Problem: doesn't solve CS problem! Reading/Setting lock not atomic!

Critical Sections — spinlocks

to make lock variable approach work, lock variable must be tested and set at same time atomically:

x86: `xchg` can atomically exchange memory content with register
- exchanges register content with memory content
- returns previous memory content of lock
⇝ implementation of critical section as *spinlock*:

```
void enter_critical_section (volatile bool *lock) {
  while (xchg(lock, 1) == 1); // lock = 1 and return old value
                              // repeat until old value != 1
}

void leave_critical_section (volatile bool *lock) {
  *lock = 0;
}
```

Advantages:
- *mutual exclusion*: only one thread can enter CS
- *progress*: only thread within CS hinders others of getting in

Disadvantages:
- *bounded waiting*: no upper bound

Spinlocks — Limitations

Congestion:
- if most times there is no thread in CS when another tries to enter, then spinlocks are very easy + efficient
- if CS is large or many threads try to enter, spinlocks might not be good choice as all threads actively wait spinning

Multicore: memory address is written at every atomic swap operation
⟶ memory is expensively kept coherent

Static Priorities (e.g., *priority inversion*): if low-priority threads hold lock it will never be able to release it, because it will never be scheduled

Spinlocks — sleep while wait

Problem: busy part of busy waiting
- wastes resources,
- stresses cache coherence protocol,
- can cause priority inversion problem

Idea:
- threads sleep on locks if occupied
- wake up threads one at a time when lock becomes free

Spinlocks — semaphore

two new syscalls operating on `int` variables:
- `wait (&s)`: if `s > 0`: `s--` and continue, otherwise let caller sleep
- `signal (&s)`: if no thread is waiting: `s++`, otherwise wake one up

initialize `s` to maximum number of threads that may enter CS
- `wait` = `enter_critical_section()`
- `signal` = `leave_critical_section()`

mutex (semaphore): semaphore initialized to 1 (only admits one thread at a time into CS)

counting semaphore: semaphore allowing more than one thread into CS at a time

Semaphore — implementation

`wait` and `signal` calls need to be carefully synchronized (otherwise *race condition* between checking and decrementing `s`)

signal loss can occur when waiting and waking threads up at same time

⇝ each semaphore has **wake-up queue**:
- *weak semaphores*: wake up random waiting thread on `signal`
- *strong semaphores*: wake up thread strictly in order which they started `wait`ing

Advantages:
- *mutual exclusion*: only one thread can enter CS for mutexes
- *progress*: only thread within CS hinders others to get in
- *bounded waiting*: strong semaphores guarantee bounded waiting

Disadvantages:
- every enter and exit of CS is syscall ⟶ slow

Fast User Space mutex

spinlock:
+ quick when wait-time is short
- waste resources when wait-time is long

semaphore:
+ efficient when wait-time is long
- syscall overhead at every operation

futex:
- userspace + kernel component
- try to get into CS with userspace spinlock
  · CS busy ⟶ use syscall to put thread to sleep
  · otherwise ⟶ enter CS with now locked spinlock completely in userspace

Summary

communication between processes/threads often needed
- *message passing*: provide explicit send/receive functions to exchange messages
- *implicitly/explicitly shared memory* between threads/processes: allows information exchange

data races: need to be taken into account when communicating

synchronization techniques:
- interlocked atomic operations
- spinlocks
- semaphores
- futexes

Synchronization and Deadlocks

Producer-Consumer Problem

Definition:
- buffer is shared between producer and consumer (LIFO)
- `count` integer keeps track of number of currently available items
- producer produces item → placed in buffer, `count++`
- buffer full → producer needs to sleep until consumer consumed an item
- consumer consumes item → remove item from buffer, `count--`
- buffer empty → consumer needs to sleep until producer produces item

Problem: *race condition* on `count`

Producer-Consumer Problem — condition variables

Solution: can be solved with mutex + 2 counting semaphores
- hard to understand
- hard to get right
- hard to transfer to other problems

condition variables: allow blocking until condition is met
- usually suitable for same problems but much easier to get right

Idea:
- new operation performs *unlock*, *sleep*, *lock* atomically
- new wake-up operation is called with lock held

→ simple mutex lock/unlock around CS + no signal loss

Pthread condition variables:
- `pthread_cond_init`: create + initialize new CV
- `pthread_cond_destroy`: destroy + free existing CV
- `pthread_cond_wait`: block waiting for signal
- `pthread_cond_timedwait`: block waiting for signal or timer
- `pthread_cond_signal` : signal another thread to wake up
- `pthread_cond_broadcast`: signal all threads to wake up

```
void producer()
{
  Item newItem;

  for(;;) // ever
  {
    newItem = produce();

    mutex_lock( &lock );
    while( count == MAX_ITEMS )
      cond_wait( &less, &lock );

    insert( newItem );
    count++;

    cond_signal( &more );
    mutex_unlock( &lock );
  }
}
```

```
void consumer()
{
  Item item;

  for(;;) // ever
  {
    mutex_lock( &lock );
    while( count == 0 )
      cond_wait( &more, &lock );

    item = remove();
    count--;

    cond_signal( &less );
    mutex_unlock( &lock );

    consume( item );
  }
}
```

Reader-Writer Problem

Problem: model access to shared data structures
- many threads compete to read/write same data
- *readers*: only read data set, not performing any updates
- *writers*: both read and write

⇝ using single mutex for read/write operations is not a good solution! (unnecessarily blocking out multiple readers while no writer is present)

Idea: locking should reflect different semantics for reading/writing
- no writing thread → multiple readers may be present
- writing thread → no other reader/writer allowed

Dining-Philosophers Problem

Definition: 5 philosophers with cyclic workflow:
- think
- get hungry
- grab one chopstick
- grab other chopstick
- eat
- put down chopsticks

Rules:
- no communication
- no atomic grabbing of both chopsticks
- no wrestling

Abstraction: models threads competing for limited number of resources **Problem**: what happens if all philosophers grab left chopstick at once?

Deadlock workarounds:
- just 4 philosophers allowed at table of 5 → *deadlock avoidance*
- odd philosophers take left chopstick first, even ones take right first → *deadlock prevention*

Deadlocks

Deadlocks can arise if all four conditions hold simultaneously:
- *mutual exclusion*: limited resource access (can only be shared with finite number of users)
- *hold and wait*: wait for next resource while already holding at least one
- *no preemption*: granted resource cannot be taken away but only handled back voluntarily
- *circular wait*: possibility of circularity in requests graph

Deadlocks — countermeasures

prevention: pro-active, make deadlocks impossible to occur

avoidance: decide on allowed actions based on a-priori knowledge

detection (*recovery*): react after deadlock happened

Deadlocks — prevention

Goal: negate at least one of the required deadlock conditions:
- *mutual exclusion*: buy more resources, split into pieces, virtualize
- *hold and wait*: get all resources en-bloque, 2-phase-locking
- *no preemption*: virtualize to make preemptable
- *circular wait*: reorder resources, prevent through partial order on resources

Deadlocks — avoidance

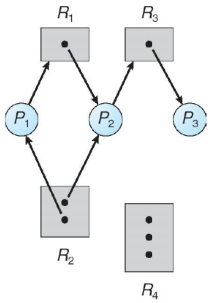
safe state: system is in safe state ⟶ no deadlocks

unsafe state: system is in unsafe state ⟶ deadlocks possible

avoidance: on every resource request decide if system stays in safe state
⟶ *resource allocation graph*

Deadlock Avoidance — resource allocation graph

principle: view system state as graph
— *processes* = round nodes
— *resources* = square nodes
— *resource instance* = dot in resource node
— *resource requests/assignments* = edges
· resource ⟶ process = resource is assigned to process
· process ⟶ resource = process is requesting resource

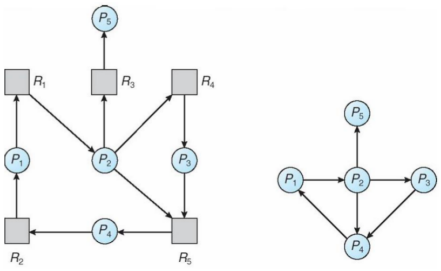


Deadlocks — detection

principle: allow system to enter deadlock ⟶ detection ⟶ recovery scheme

wait-for graph (WFG):
— *processes* = nodes
— *wait-for relationship* = edges

periodically invoke algorithm searching for cycle in graph
⇝ cycle exists ⟶ deadlock exists



Deadlocks — recovery

process termination:
— *all*: abort all deadlocked processes
— *selective*: abort one process at a time until deadlock is eliminated

termination order: in which order should processes be aborted?
— process priority
— how long already computed? how much longer for completion?
— amount of resources used
— amount of resources needed for completion
— how many processes will need to be terminated
— interactive or batch?

resource preemption:
— *victim selection*: minimize cost
— *rollback*: perform periodic snapshots, abort process to preempt resources ⟶ restart from last safe state
— *starvation*: same process may always be picked as victim ⇝ include rollback count in cost factor

Summary

classical synchronization problems: model synchronization problems occurring in reality
— *producer-consumer*: shared use of buffers/queues
— *reader-writer*: shared access to data structures
— *dining philosophers*: competition for limited resources

such synchronization problems occur very often when programming operating systems

parallelism: introduced by multiple processors + multiprogramming, needs to be considered carefully when writing OS

Memory Management Hardware

Main Memory

main memory + registers = only storage that CPU can access directly

before run: program must be
- brought into memory from background storage
- placed within a process' address space

Earlier: computers had no memory abstraction
⟶ programs accessed physical memory directly

multiple processes can be run concurrently even without memory abstraction (using swapping, relocation)

Swapping

Principle:
- *roll-out*: save program's state on background storage
- *roll-in*: replace program state with another program's state

Advantages:
- only needs hardware support to protect kernel, not to protect processes from one another

Disadvantages:
- *very slow*: major part of swap time is transfer time
- *no parallelism*: only one process runs at a time, owns entire physical address space

Overlays

Problem: what if process needs more memory than available?
⟶ need to partition program manually

Static Relocation

= OS adds fixed offset to every address in a program when loading + creating process

same address space for every process
⟶ *no protection*: every program sees + can access every address!

Shared Physical Memory — Goals

Protection:
- bug in one process must not corrupt memory in another
- do not allow processes to observe other processes' memory

Transparency:
- process should not require particular physical memory addresses
- processes should not be able to use large amounts of contiguous memory

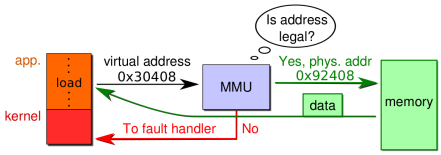
Resource Exhaustion:
- allow that sum of sizes of all processes is greater than physical memory

Memory Management Unit

need hardware support to achieve safe + secure protection

Goal: hardware maps virtual to physical address

Usage: user program deals with virtual addresses, never sees real addresses



MMU — base and limit registers

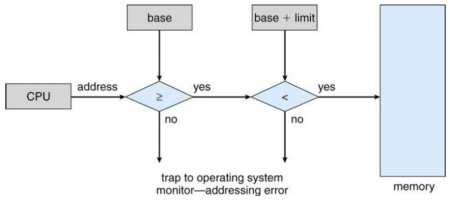
Idea: provide protection + dynamic relocation in MMU
→ introduce special *base* and *limit* registers (e.g., Cray-1)

Usage: on every load/store the MMU
- checks if virtual address $\geq$ `base`
- checks if virtual address $<$ `base` + `limit`
- use virtual address as physical address in memory

Protection: OS needs to be protected from processes
- main memory split in two partitions (low = OS, high = user processes)
- OS can access all process partitions (e.g., to copy syscall parameters)
- MMU denies processes access to OS memory

Advantages:
- straight forward to implement MMU
- very quick at run-time

Disadvantages:
- how to grow process' address space?
- how to share code/data?



MMU — segmentation

Solution to base + limit: use multiple base + limit register pairs *per process*
→ private + public segments

Advantages:
- data/code sharing between processes possible without compromising confidentiality
- process does not need large contiguous physical memory area → easy placement
- process does not need to be entirely in memory → memory overcommitment ok

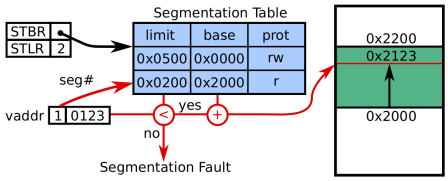
Disadvantages:
- segments need to be kept contiguous in physical memory
- *fragmentation* of physical memory

Segmentation — architecture

virtual address = [segment #, offset]

each process has *segment table*, maps virtual address to physical address in memory
- *base*: starting physical address where segment resides in memory
- *limit*: length of segment
- *protection*: access restriction (read/write) for safe sharing

MMU has two registers that identify current address space
- *segment-table base register* (STBR): points to segment table location of current process
- *segment-table length register* (STLR): indicates number of segments used by process



External Fragmentation

Fragmentation = inability to use free memory

External Fragmentation = sum of free memory satisfies requested amount of memory, but is not contiguous

Compaction: reduce external fragmentation
- close gaps by moving allocated memory in one direction
- only possible if relocation is dynamic, can be done at execution time
- *problem*: expensive! Need to halt process while moving data and updating tables
  → caches need to be reloaded, which should be avoided

MMU — paging

Principle: divide physical memory into fixed-size blocks (*page frames*)
- size = 2^n Bytes (typically 4KiB, 2MiB, 4MiB)

Virtual Memory: divided into same-sized blocks (*pages*)

Page Table: managed by OS, stores mappings between *virtual page numbers* (vpn) and *page frame numbers* (pfn) for each AS

OS tracks all free frames, modifies page tables as needed

Present Bit (in page table): indicates that virtual page is currently mapped to physical memory

if process issues instruction to access unmapped virtual address, MMU calls OS to bring in the data (*page fault*)

MMU — address translation scheme

Virtual address: divided into
- *virtual page number*: page table index containing base address of each page in physical memory
- *page offset*: concatenated with base address results in physical address

MMU — hierarchical page table

Problem: need to keep complete page table in memory for every address space

Idea: not needing complete table, most virtual addresses unused by process
→ subdivide virtual address further into multiple page indexes p_n forming *hierarchical page table*

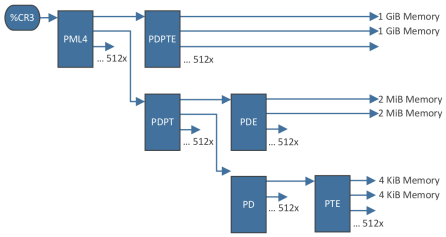
Hierarchical Page Table — x86-64

long mode: 4-level hierarchical page table

page directory base register (control register 3, `%CR3`) stores starting physical address of *first level page table*

address-space hierarchy: following page-table hierarchy for every address space:
- page map level 4 (PML4)
- page directory pointers table (PDPT)
- page directory (PD)
- page table entry (PTE)

per level: table can either point to *directory* in next hierarchy level or to *entry* containing actual mapping data



Page Table Entry — content

valid bit (*present bit*): whether page is currently available in memory or needs to be brought in by OS via *page fault*

page frame number: if page is present: physical address where page is currently located

write bit: whether or not page may be written to (may cause *page fault*)

caching: whether or not page should be cached at all (and with which policy)

accessed bit: set by MMU if page was touched since bit was last cleared by OS

dirty bit: set by MMU if page was modified since bit was last cleared by OS

Paging — OS involvement

OS performs all operations that require semantic knowledge

page allocation (bringing data into memory): OS needs to find free frame for new pages and set up mapping in page table of affected address space

page replacement: when all page frames are used, OS needs to evict pages from memory

context switching: OS sets MMU's base register (`%CR3` on x86) to point to page hierarchy of next process's address space

MMU — internal fragmentation

paging: eliminates external fragmentation

problem: internal fragmentation
- memory can only be allocated in page frame sizes
- allocated virtual memory area will generally not end at page boundary
- ⇝ unused rest of last allocated page is lost!

MMU — page size trade-offs

fragmentation:
- *larger pages* ⟶ more memory wasted (internal fragmentation) per allocation
- *smaller pages* ⟶ only half a page wasted per allocation on average

table size:
- *larger pages* ⟶ fewer bits needed for `pfn` (more bits in offset), fewer PTEs
- *smaller pages* ⟶ more + larger PTEs

I/O:
- *larger pages* ⟶ more data needs to be loaded from dist to make page valid
- *smaller pages* ⟶ need to trap OS more often when loading large program

Summary

need to place processes in memory to run

want to place multiple processes in memory at same time to run concurrently/parallel

virtual memory: enables protection, transparency, overcommitment
— *trade-off* extra hardware (MMU) to translate addresses at every load/store

MMU types: base + limit, segmentation, paging

paging: supported by all contemporary MMUs, favorite of most OS

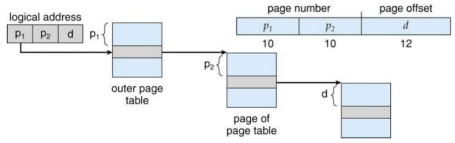
Paging

Hierarchical Page Table — two-level page table

layout: on 32-bit machine with 4KiB pages divide virtual address into
— *page number* (p): 20 bits
— *page offset* (d): 12 bits

table paging: table can be paged to save memory – subdivide vpn:
— index in *page directory* (p_1): 10 bits
— index in *page table entry* (p_2): 10 bits

for ranges of 1024 invalid pages, reset present bit in page directory
⟶ save space of second-level page table



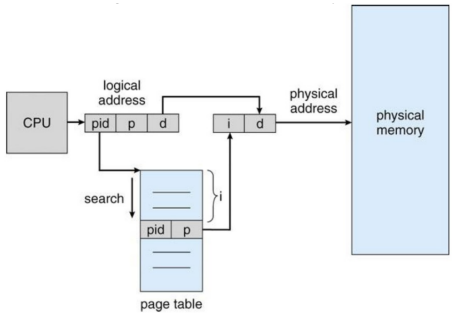
Linear Inverted Page Table

Problem: large AS (64 bit) but only few mapped virtual addresses
⟶ much memory wasted on page tables
⟶ lookup slow due to many levels of hierarchy

Idea: invert page table mapping
— map physical frame to virtual page instead of other way around
— single page table for *all processes* (exactly one table per system)
— one page table entry for each physical page frame

Advantage: less overhead for page table meta data

Disadvantage: increases time needed to search table when page reference occurs



Hashed Inverted Page Table

hash anchor table: limits search to at most a few page-table entries

Translation Lookaside Buffer — motivation

naive paging is slow:
— every load/store requires multiple memory references
— 4-level hierarchy: 5 memory references for every load/store
(4 page directory/table references, 1 data access)

Idea: add cache that stores recent memory translations
— *translation lookaside buffer* (TLB) maps [vpn] to [pfn, protection]
— typically 4-way to fully associative hardware cache in MMU
— typically 64-2000 entries
— typically 95%-99% hit rate

TLB — operation

on every load/store:
- check if translation result is cached in TLB (*TLB hit*)
- otherwise walk page tables, insert result into TLB (*TLB miss*)

Quick: can compare many TLB entries in parallel in hardware

TLB — TLB miss

Process:
- evict entry from TLB on TLB miss
- load entry for missing virtual address into TLB

Variants: *software-managed* and *hardware managed*

software-managed TLB:
- OS receives *TLB miss exception*
- OS decides which entry to evict (drop) from TLB
- OS generally walks page tables in software to fill new TLB entry
- TLB entry format specified in *instruction set architecture* (ISA)

hardware-managed TLB:
- evict TLB entry based on hardware-encoded policy
- walk page table in hardware → resolve address mapping

TLB — address space identifiers

Problem: vpn dependent on AS
- vpns in different AS can map to different pfns
- → need to clear TLB on AS switch

Idea: solve vpn ambiguity with additional identifiers in TLB

ASID: TLB has *address space identifier* (ASID) in every entry
- map [vpn, ASID] to [pfn, protection]
- → avoids TLB flush at every address-space switch
- → less TLB misses: some TLB entries still present from last time process ran

TLB — reach

= amount of memory accessible with TLB hits:

$$\text{TLB reach} = \text{TLB size} * \text{page size}$$

ideally: working set of each process is stored in TLB (otherwise high degree of TLB misses)

increase page size:
- \+ fewer TLB entries per memory needed
- − increase internal fragmentation

multiple page sizes:
- \+ allows applications that map larger memory areas to increase TLB coverage with minimal fragmentation increase

increase TLB size:
- − expensive

TLB — effective access time

associative lookup: takes τ time units (e.g., $\tau = 1$ns)

memory cycle: takes μ time units (e.g., $\mu = 100$ns)

TLB hit ratio α: percentage of all memory accesses with cached translation (e.g., $\alpha = 99\%$)

effective access time (EAT) for linear page table without cache:

$$\text{EAT} = (\tau + \mu)\alpha + (\tau + 2\mu)(1 - \alpha) = \tau + 2\mu - \mu\alpha$$

Summary

page tables communicate between OS and MMU hardware
- how virtual addresses in each address space translate to physical addresses
- which kind of accesses the MMU should allow/signal to the OS

different page table layouts have been developed
- linear page table
- hierarchical page tables
- inverted page tables
- hashed page tables

performing page table lookups for every memory access significantly slows down execution time of programs
- translation lookaside buffer (TLB) caches previously performed page table lookups
- typical TLBs cover $95\% - 99\%$ of all translations

Caching

Caching — motivation

memory (RAM) needs to be managed carefully

Ideal properties: large, fast, nonvolatile, cheap

Real memory: trade-offs

Caching — cache misses

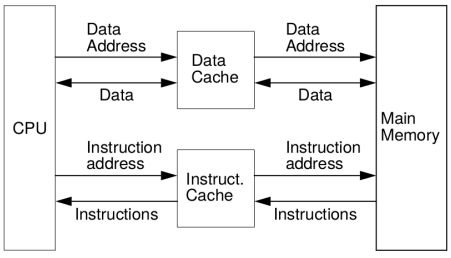
compulsory miss:
- cold start, first reference
- data block was not cached before

capacity miss:
- not all required data fits into cache
- accessed data previously evicted to make room for different data

conflict miss:
- collision, interference
- depending on cache organization, data items interfere with each other
- fully associative caches are not prone to conflict misses

Caching — Harvard architecture

principle: separate buffer memory for data and instructions



Caching — write/replacement policies

cache hit:
- *write-through*: main memory always up-to-date, writes might be slow
- *write-back*: data written only to cache, main memory temporarily inconsistent

cache miss:
- *write-allocate*: data read from main memory to cache, write performed afterwards
- *write-to-memory*: modification is performed only in main memory

Cache Design Parameters

size + set size: small cache → set-associative implementation with large sets

line length: spatial locality → long cache lines

write policy: temporal locality → write-back

replacement policy

tagging/indexing: virtual or physical addresses

Caching — problems

ambiguity problem: same virtual addresses point to different physical addresses at different times

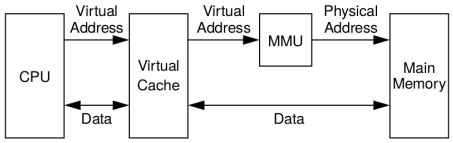
alias problem: different virtual addresses point to same physical memory location

Caching — virtually indexed, virtually tagged

operations:
- *context switch*: cache must be invalidated (and written back if write-back is used)
- *fork*: child needs complete copy of parent's address space
- *exec*: invalidate cache, no write-back necessary
- *exit*: flush cache
- *brk/sbrk*: growing = nothing, shrinking = (selective) cache invalidations

shared memory/memory-mapped files: alias problem!
- disallow, do not cache
- only allow addresses mapping to same cache line (if using direct-mapped write-allocate cache)
- each frame accessible from exactly one virtual address at any time → alias page invalidation

I/O:
- *buffered I/O*: no problems
- *unbuffered I/O*:
  - write: information may still be in cache → write back before I/O starts
  - read: cache must be invalidated



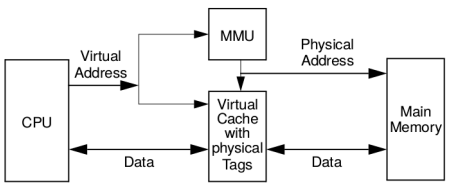
Caching — virtually indexed, physically tagged

usage: often used as first-level cache

management:
- no ambiguities
- no cache flush/context switch
- shared memory/memory mapped files: virtual starting addresses must be mapped to same cache line
- I/O: cache flush required

conflicts: data structures with address distance = multiple of cache size are mapped to same line

runtime properties:
- *cache flush*: avoidable most times (fast context switches, interrupt-handling, syscalls)
- *deferred write-back after context switch*:
  - avoids write accesses → performance gain
  - variable execution time caused by compulsory misses
- *dynamic memory management*: causes variable execution times through conflict misses
- *multiprocessor systems*: problematic with shared memory — which line should be invalidated?
  - cache size is small multiple of page size (1-4)
  - requires to only invalidate/flush 1-4 cache lines by cache coherency HW



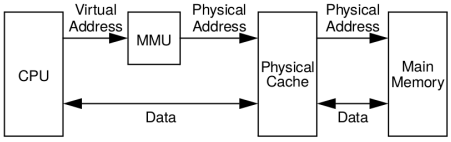
Caching — physically indexed, physically tagged

Advantages:
- completely transparent to processor
- no performance-critical system support required (including I/O)
- SMPs with shared memory can use coherency protocol implemented in hardware

random allocation conflicts:
- page conflicts caused by random allocation of physical memory
- contiguous virtual memory normally mapped to arbitrary free physical pages

random coloring conflicts: consequences of random page coloring:
- cache conflicts
- cache only partially used
- significant runtime variations

conflict mitigation:
- sequential page colors for individual memory segments
- *cache partitioning*: divide physical memory in disjoint subsets, all pages of subset are mapped to same cache partition



Page Faults

Page Faults — handling

Cause: access to page currently not present in main memory
→ exception, invoking OS

Process:
- OS checks access validity (requiring additional info)
- get empty frame
- load contents of requested page from disk into frame
- adapt page table
- set present bit of respective entry
- restart instruction causing page fault

Page Faults — latency

fault rate $0 \leq p \leq 1$
- $p = 0$: no page faults
- $p = 1$: every reference leads to page fault

effective access time (EAT):
$\text{EAT} = (1 - p) * \text{memory access} + p * (\text{PF overhead} + \text{PF service time} + \text{restart overhead})$

Page Faults — performance impact

memory access time: 200ns

average page fault service time: 8ms

$\rightsquigarrow$ 1:1000 access-page-fault-rate → EAT $= 8.2\mu s \Rightarrow$ *slowdown by factor 40*!

Page Faults — challenges

what to eject?
- how to allocate frames among processes?
- which particular process's pages to keep in memory?
- see *page frame allocation*

what to fetch?
- what if block size $\neq$ page size?
- just one page needed? prefetch more?

process resumption?
- — need to save state + resume
- — process might have been in middle of instruction

Page Faults — what to fetch?

bring in page causing fault

pre-fetch sourrounding pages?
- — reading two disk blocks is approximately as fast as reading one
- — as long as there is no track/head switch, seek (disk) time dominates
- — application exhibits spatial locality = big win

pre-zero pages?
- — don't want to leak information between processes
- — need 0-filled pages for stack, heap, .bss, …
- — *zero on demand*?
- — keep pool of 0-pages filled in background when CPU is idle?

Page Faults — process resumption?

hardware provides info about page fault
(intel: `%cr2` contains faulting virtual address)

context: OS needs to figure out fault context:
- — read or write?
- — instruction fetch?
- — user access to kernel memory?

idempotent instructions: easy:
- — re-do load/store instructions
- — re-execute instructions accessing only one address

complex instructions: must be re-started
- — some CISC instructions are hard to restart (e.g., block move of overlapping areas)
- — *solutions*:
  - · touch relevant pages before operation starts
  - · keep modified data in registers $\rightarrow$ page faults can't take place
  - · design ISA such that complex operations can execute partially $\rightarrow$ consistent page fault state

Memory-Mapped Files — other issues

I/O mapping: mapping disk block to page in memory allows file I/O to be treated as routing memory
- — *initial*: read page-sized portion of file from file system to physical page
- — *subsequent read/write*: treated as ordinary memory access
- $\rightarrow$ *simplifies* file access, file I/O through memory instead of syscalls
- $\rightarrow$ *memory-file sharing*: several processes can map to same file

Shared Data Segments

implementation:
- — temporary, asynchronous memory-mapped files
- — shared pages (with allocated space on backing store)

copy on write (COW):
- — allows both parent and child process to initially share same memory pages
- — only modified pages are copied $\rightarrow$ more efficient process creation

Page Frame Allocation — local vs. global

global: all frames considered for replacement
- — does not consider page ownership
- — one process cannot get another process's frame
- — does not protect process from a process that hogs all memory

local: only frames of faulting process are considered for replacement
- — isolates processes/users
- — separately determine how many frames each process gets

Fixed Allocation — equal vs. proportional

equal: all processes get same amount of frames

proportional: allocate according to process size
$s_i :=$ size of process p_i, $S := \sum s_i$, $m :=$ total number of frames
$\Rightarrow a_i := \frac{s_i}{S} m$ allocation for p_i

Fixed Allocation — priority allocation

= proportional allocation scheme using priorities rather than size

on page fault of P_i:
- — select one of its frames for replacement or
- — select frame from process with lower priority

Memory Locality

problem: background storage much slower than memory
- — paging extends memory size using background storage
- — *goal*: run near memory speed, not near background storage speed

Pareto principle: applies to working sets of processes
- — 10% of memory gets 90% of references
- — *goal*: keep those 10% in memory, rest on disk
- — *problem*: how to identify those 10%?

Thrashing

problem: system is busy swapping pages in and out
- — each time one page is brought in, another page, whose contents will soon matter, is thrown out
- — *effect*: low CPU utilization, processes wait for pages to be fetched from disk
- — *consequence*: OS thinks that it needs higher degree of multiprogramming

reasons:
- — *no temporal locality* of access pattern — process doesn't follow Pareto principle
- — *too much multiprogramming*: each process fits individually, but too many for system
- — *memory too small* to hold hot memory of a single process (the 10%)
- — *bad page replacement policy*

Working-Set Model

$\Delta :=$ working-set window (fixed number of page references; e.g., 10000 instructions)

$\text{WSS}_i :=$ working set of process P_i
- — total number of pages referenced in most recent Δ (varies in time)
- — $\Delta \begin{cases} \text{too small} & \Rightarrow \text{ will not encompass entire locality} \\ \text{too large} & \Rightarrow \text{ will encompass several localities} \\ = \infty & \Rightarrow \text{ will encompass entire program} \end{cases}$

$D := \sum \text{WSS}_i$ = total demand for frames
- — $D > m \rightsquigarrow$ **thrashing**
- $\rightarrow D > m \Rightarrow$ suspend a process

Working Set — keeping track

perfect: replace page that is referenced furthest in the future (*oracle*)

idea: predict future from past
- — record page references from past and extrapolate into future
- — *problem*: too expensive to make ordered list of all page references at runtime

idea: sacrifice precision for speed
- — MMU sets *reference bit* in respective page table entry every time a page is referenced
- — set timer to scan all page table entries for reference bits

Page Fault Frequency — allocation scheme

goal: establish acceptable page fault rate
- — *actual rate too low* $\rightarrow$ give frames to other process
- — *actual rate too high* $\rightarrow$ allocate more frames to process

Page Fetch Policy — demand paging

idea: only transfer pages raising page faults

advantages:
- — only transfer what is needed
- — less memory needed by process ⟶ higher multiprogramming degree possible

disadvantages:
- — many initial page faults when task starts
- — more I/O operations ⟶ more I/O overhead

Page Fetch Policy — pre-paging

idea: speculatively transfer pages to RAM
- — at every page fault: speculate what else should be loaded
- — e.g., load entire text section when process starts

advantage: improves disk I/O throughput

disadvantages:
- — wastes I/O bandwidth if page is never used
- — can destroy working set of other processes in case of page stealing

Summary

paging simulates a memory size of the size of the virtual memory

when pages are filled via page faults, OS needs to answer some questions:
- — what to eject?
- — what to fetch?
- — how to resume process?

different strategies to allocate frames and replace pages:
- — local vs. global allocation
- — fixed vs. proportional vs. priority allocation

thrashing must be prevented by taking working sets of active processes into account

Page Replacement Policies

Page Replacement — naive

step 1: save/clear victim page:
- — drop page if fetched from disk and clean
- — *dirty*: write back modifications if from disk and dirty (unless `MAP_COPY`)
- — *non-dirty*: write page file/swap partition otherwise (e.g., stack, heap memory)

step 2: unmap page from old AS: invalidate PTE, flush cache

step 3: prepare new page: null page or load new contents

step 4: map page frame into new AS: invalidate PTE, flush cache

Page Replacement — buffering

problem: naive page replacement encompasses two I/O transfers
⟶ both operations block page fault from completing

goal: reduce I/O from critical page fault path to speed up page faults

idea: keep pool of free page frames (*pre-cleaning*):
- — *on page fault*: use page frame from free pool
- — *cleaning*: daemon cleans, reclaims and scrubs pages for free pool in background

⟶ smooths out I/O, speeds up paging significantly

remaining problem: which pages to select as victims?
- — *goal*: identify page that has left working set of its processes, add to free pool
- — *success metric*: low overall page fault rate

Page Replacement — FIFO

idea: evict oldest fetched page in system

Belady's Anomaly: using FIFO, for every number n of page frames you can construct a reference string that performs worse with $n + 1$ frames
⟶ with FIFO it is possible to get more page faults with more page frames!

Page Replacement — oracle

= optimal replacement strategy: replace page whose next reference is furthest in future

problem: future unpredictable

however: good metric to check how well other algorithms perform

Page Replacement — LRU

goal: approximate oracle page replacement

idea: past often predicts future well

assumption: page used furthest in past is used furthest in future

cycle counter implementation:
- — have MMU write CPU's time stamp counter to PTE on every access
- — *page fault*: scan all PTEs to find oldest counter value
- — *advantage*: cheap at access if done in HW
- — *disadvantage*: memory traffic for scanning

stack implementation:
- — keep doubly linked list of all page frames
- — move each referenced page to tail of list
- — *advantage*: can find replacement victim in $O(1)$
- — *disadvantage*: need to change 6 pointers at every access

⇝ **no silver bullet**:
- — *observation*: predicting future based on past is not precise
- — *conclusion*: relax requirements — maybe perfect LRU isn't needed? ⇒ approximate LRU

LRU Approximation — clock page replacement

aka *second chance page replacement*

precondition: MMU sets reference bit in PTE
- — supported natively by most hardware
- — can easily emulate in systems with software managed TLB (e.g., MIPS)

store: keep all pages in circular FIFO list

searching for victim: scan pages in FIFO's order
- — if reference bit = 0 ⟶ use page as victim and advance
- — if reference bit = 1 ⟶ set to 0, continue scanning

problem: large memory ⟶ most pages referenced before scanned
- — *solution*: use 2 arms, leading arm clears reference bit, trailing arm selects victim

Replacement Strategies — other

random eviction: pick random victim
- — dirt simple
- — not overly horrible in reality

larger counter: use n-bit reference counter instead of reference bit
- — *least frequently used*: rarely used page not in a working set ⟶ replace page with smallest count
- — *most frequently used*: page with smallest count probably just brought in ⟶ replace page with largest count
- — neither LFU nor MDU are common (no such hardware, not that great)

Summary

victim page frame needs to be selected by OS when handling page faults
- evicting page frame after page fault happens = not a good idea
- page buffering keeps eviction out of critical path

different victim selection policies exist
- FIFO ⟶ Belady's Anomaly
- Oracle ⟶ cannot predict the future
- Random ⟶ unpredictable, never great but rarely very bad
- LRU ⟶ hard to implement efficiently

Memory Allocation

Memory Allocation — dynamic

= allocate + free memory chunks of arbitrary size at arbitrary points in time
- almost every program uses it (heap)
- don't have to statically specify complex data structures
- can have data grow as function of input size
- kernel itself uses dynamic memory allocation for its data structures

implementation: has huge impact on performance, both in user and kernel space

fact: it is impossible to construct memory allocator that always performs well
⟶ need to understand trade-offs to pick good allocation strategy

Dynamic Memory Allocation — principle

initial: pool of free memory

tasks:
- satisfy arbitrary `allocate` + `free` requests from pool
- track which parts are in use/are free

restrictions:
- cannot control order/number of requests
- cannot move allocated regions ⟶ fragmentation = core problem!

Dynamic Memory Allocation — bitmap

idea:
- divide memory in allocation units of fixed size
- use bitmap to keep track if allocated (1) or free (0)

problem: needs additional data structure to store allocation length (otherwise cannot infer whether two adjacent allocations belong together or not from bitmap)

Dynamic Memory Allocation — list

method 1: use one list-node for each allocated data
- *extra space* needed for list
- allocation lengths already stored

method 2: use one list-node for each unallocated data
- can keep list in unallocated area (store size of free area + pointer to next free area in free area)
- *additional data structure* needed to store allocation lengths
- can search for free space with low overhead

method 3: both

Dynamic Memory Allocation — problems

fragmentation is hard to handle

factors needed for fragmentation to occur:
- *different lifetimes*
- *different sizes*
- *inability to relocate previous allocations*

all fragmentation factors present in dynamic memory allocators!

Allocation — best fit vs. worst fit

idea: keep large free memory chunks together for larger allocation requests that may arrive later

best-fit: allocate smallest free block large enough to store allocation request
- must search entire list
- *problem*: sawdust — remainder so small that over time left with unusable sawdust everywhere
- *idea*: minimize sawdust by turning strategy around

worst-fit: allocate largest free block
- must search entire list
- *reality*: worse fragmentation than best-fit

Allocation — first fit

idea: if fragmentation occurs with best and worst fit, optimize for allocation speed

principle: allocate first hole big enough
- fastest allocation policy
- produced leftover holes of variable size
- *reality*: almost as good as best-fit

First Fit — variants

first-fit sorted by address order

LIFO first-fit

next fit

Allocation — buddy allocator

idea: allocate memory in powers of 2
- all chunks have fixed 2^n-size ⟶ allocation request rounded up to next-higher power of 2
- all chunks naturally aligned

no sufficiently small block available:
- select larger available chunk, split into two same-sized buddies
- continue until appropriately sized chunk is available

two buddies both free (2^n): merge to 2^{n+1}-chunk

Real Program Patterns

ramps: accumulate data monotonically over time

peaks: allocate many objects, use briefly, then free all

plateaus: allocate many objects, use for long time

Allocation — slabs

kernel often allocates/frees memory for few, specific data objects of fixed size

slab: multiple pages of contiguous physical memory
- linux: uses buddy allocator as underlying allocator for slabs

cache: one or multiple slabs
- stores only one kind of object (fixed size)

Summary

dynamic memory means allocating and freeing memory chunks of different sizes at any time

impossible to construct memory allocator that always performs well

typical dynamic memory data structures:
- bitmaps
- lists

simple, well-performing allocation strategies:
- best-fit
- first-fit

advanced strategies:
- buddy-allocator
- slab-allocator

Secondary Storage

Secondary Storage — structure

hard disk drives

solid state drive

RAID structure

tertiary storage devices (DVD, magnetic tape)

Hard Disk Drives — anatomy

stack of magnetic platters

disk arms contain disk heads per recording surface, read/write to platters

storage:
- platters divided into concentric *tracks*
- *cylinder*: stack of tracks of fixed radius
- tracks of fixed radius divided into *sectors*

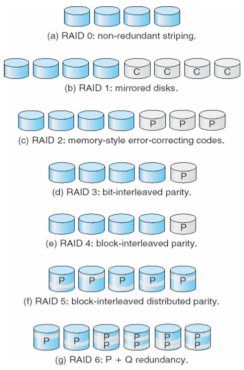
Flash Memory

advantages:
- solid state
- lower power consumption/heat
- no mechanical seek

disadvantages:
- limited number of overwrites
- limited durability

RAID

idea: improve performance + reliability of storage system by storing redundant data



File Systems

File Systems — motivation

goal: enable storing of large data amounts
- store data/program consistently + persistently
- easily look up previously stored data/program

file types:
- *data* (numeric, character, binary)
- *program*

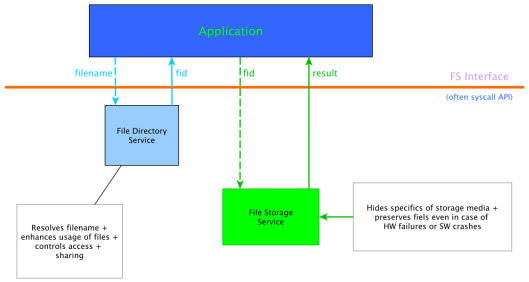
File Systems — overview

OS may support multiple file systems

namespace: all file systems typically bound into single namespace (often hierarchical, rooted tree)

Files – abstract operations

file: abstract data type/object, offering
- `create`, `write`, `read`,
- `reposition` (within file),
- `delete`, `truncate`,
- `open`(F_i) (search directory structure on disk for entry F_i, move meta data to memory),
- `close`(F_i) (move cached meta data of entry F_i in memory to directory structure on disk)



File Management — goals

provide convenient file naming scheme

provide uniform I/O support for variety of storage device types

provide standardized set of I/O interface functions

minimize/eliminate loss/corruption of data

provide I/O support + access control for multiple users

enhance system administration (e.g., backup)

provide acceptable performance

File Management — open files

several meta data is needed to manage open files

file pointer: pointer to last read/write location, per process that has file opened

access rights: per-process access mode information

file-open count: counter of number of times a file is opened (to allow removal of data from open-file table when last process closes)

disk location: cache of data access information

File Access

strictly sequential (early systems):
- read all bytes/records from beginning
- cannot jump round, could only rewind
- sufficient as long as storage was a tape

random access (current systems):
- bytes/records read in any order
- essential for database systems

Directories — goals

naming: convenient to users
- two users can have same name for different files
- same file can have several different names

grouping: logical grouping of files by properties

efficiency: fast operations

Files — sharing

issues:
- efficiently access to same file?
- how to determine access rights?
- management of concurrent accesses?

access rights:
- *none*: existence unknown to user, user cannot read directory containing file
- *knowledge*: user can only determine existence and file ownership
- *execution*: user can load + execute program, but can not copy it
- *reading*: user can read file (includes copying + execution)
- *appending*: user can only add data to file, but cannot modify/delete data in file
- *updating*: user can modify + delete + add to file (includes creating + removing all data)
- *change protection*: user can change access rights granted to other users
- *deletion*: user can delete file
- *owner*: all previous rights + rights granting

concurrent access:
- *application locking*: application can lock entire file or individual records for updating
- *exclusive vs. shared*: writer lock vs. multiple readers allowed
- *mandatory vs. advisory*: access denied depending on locks vs. process can decide what to do

File System Implementation

Disk Structure

partitions: disk can be subdivided into partitions

raw usage: disks/partitions can be used raw (unformatted) or formatted with file system

volume: entry containing FS
- tracks that file system's info is in device directory or volume table of contents

FS diversity: there are general purpose and special purpose FS

File Systems — logical vs. physical

logical: can consist of different physical file systems

placement: file system can be mounted at any place within another file system

mounted local root: bit in i-node of local root in mounted file system identifies this directory as mount point

File Systems — layers

layer 5: applications

layer 4: logical file system

layer 3: file-organization module

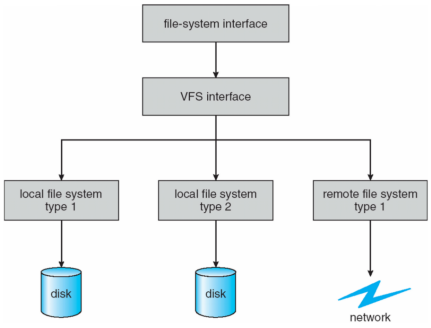
layer 2: basic file system

layer 1: I/O control

layer 0: devices

File Systems — virtual

principle: provide object-oriented way of implementing file systems
- same API used for different file system types



Files — implementation

meta data must be tracked:
- which logical block belongs to which file?
- block order?
- which blocks are free for next allocation?

block identification: blocks on disk must be identified by FS (given logical region of file)
→ meta data needed in *file allocation table*, *directory* and *inode*

block management: creating/updating files might imply allocating new/modifying old disk blocks

Allocation — policies

preallocation:
- *problem*: need to know maximum file size at creation time
- often difficult to reliably estimate maximum file size
- users tend to overestimate file size to avoid running out of space

dynamic allocation: allocate in pieces as needed

Allocation — fragment size

extremes:
- fragment size = length of file
- fragment size = smallest disk block size (= sector size)

trade-offs:
- *contiguity*: speedup for sequential accesses
- *small fragments*: larger tables needed to manage free storage and file access
- *large fragments*: improve data transfer
- *fixed-size fragments*: simplifies space reallocation
- *variable-size fragments*: minimizes internal fragmentation, can lead to external fragmentation

Allocation — file space

contiguous

chained

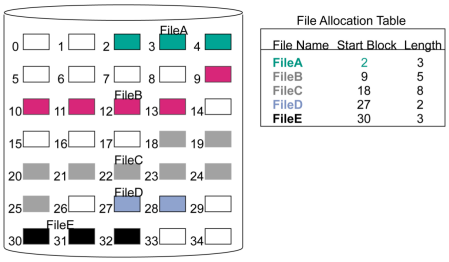
indexed:
- fixed block fragments
- variable block fragments

characteristic	contiguous	chained	indexed	
preallocation?	necessary	possible	possible	
fixed or variable size fragment?	variable	fixed	fixed	variable
fragment size	large	small	small	medium
allocation frequency	once	low to high	high	low
time to allocate	medium	long	short	medium
file allocation table size	one entry	one entry	large	medium

Allocation — contiguous

principle: array of n contiguous logical blocks reserved per file (to be created)

periodic compaction: overcome external fragmentation

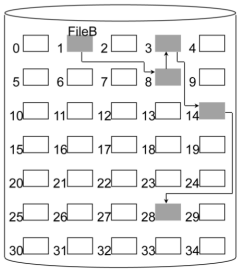


File Allocation Table

File Name	Start Block	Length
FileA	2	3
FileB	9	5
FileC	18	8
FileD	27	2
FileE	30	3

Allocation — chained

principle: linked list of logical blocks per file
— FAT or directory contains address of first file block
→ *no external fragmentation*: any free block can be added to chain

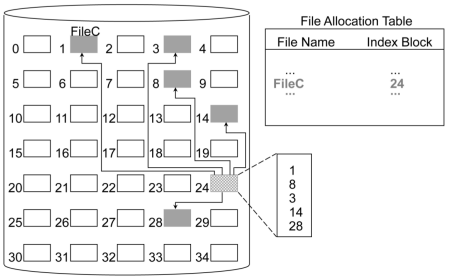


File Allocation Table

File Name	Start Block	Length
FileB	1	5

Allocation — indexed

principle: FAT contains one-level index table per file
— *generalization*: n-level index table
— index has one entry for allocated file block
— FAT contains block number for index



File Allocation Table

File Name	Index Block
FileC	24

Directories — implementation

simple directory (MS-DOS):
— fixed-size entries
— disk addresses + attributes in directory entry

i-node reference directory (UNIX):
— entry refers to i-node containing attributes

Disk Blocks — buffering

buffering: disk blocks buffered in main memory

access: buffer access done via hash table
— blocks with same hash value are chained together

replacement: LRU

management: free buffer is managed via doubly-linked list

File Systems — journaling

principle: record each update to file system as *transaction*
— written to log

committed transaction = written to log
→ *problem*: file system may not yet be updated

writing transactions from log to FS is asynchronous

modifying FS → transaction removed from log

crash of file system → remaining transactions in log must still be performed

File Systems — log-structured

principle: use disk as circular buffer
— write all updated (including i-nodes, meta data and data) to end of log

buffering: all writes initially buffered in memory

writing: periodically write within 1 segment (1 MB)

opening: locate i-node, find blocks

clearing: clear all data from other end, no longer used

I/O Systems

Device Management — objectives

abstraction from details of physical devices

uniform naming that does not depend on hardware details

serialization of I/O operations by concurrent applications

protection of standard-devices against unauthorized accesses

buffering if data from/to device cannot be stored in final destination

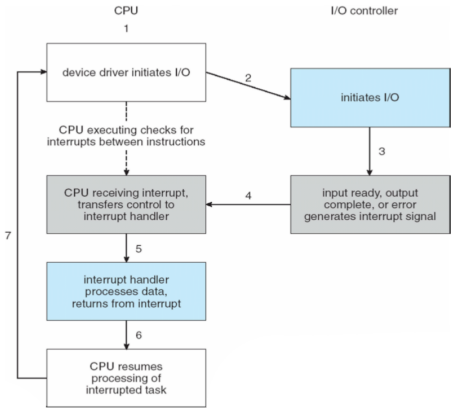
error handling of sporadic device errors

virtualizing physical devices via memory + time multiplexing

Device Management — techniques

programmed I/O:
— thread is busy-waiting for I/O operation to complete → CPU cannot be used elsewhere
— kernel is *polling* state of I/O device (command-ready, busy, error)

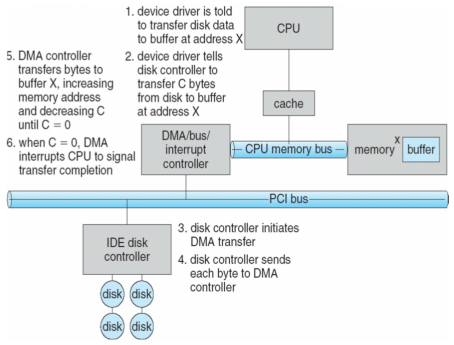
interrupt-driven I/O:
— I/O command is issued
— processor continues executing instructions
— I/O device sends interrupt when command is done



direct memory access (DMA):

- DMA module controls exchange of data between main memory and I/O device
- processor interrupted after entire block has been transferred

→ bypasses CPU to transfer data directly between I/O device and memory



Kernel I/O Subsystem

scheduling: order I/O requests in per-device queues

buffering: store data in memory while transferring between devices

error handling: recover from read/availability/write errors

protection: protect from accidental/purposeful disruptions

spooling: hold output to device if device is slow (e.g., printer)

reservation: provide exclusive access for process

Device Drivers

jobs:

- *translate* user request through device-independent standard interface
- *initialize* hardware at boot time
- *shut down* hardware

Device Buffering

reasons:

- without buffering threads must wait for I/O to complete before proceeding
- pages must remain in main memory during physical I/O

version 1 — block-oriented:

- information is stored in fixed-size blocks
- transfers are made a block at a time
- used for disks/tapes

version 2 — stream-oriented:

- transfer information as byte stream
- used for keyboard, terminals, … (most things that is not secondary storage)

Buffering — user level

principle: task specifies memory buffer where incoming data is placed

issues:

- what happens if buffer is currently paged out to disk? → data loss
- additional problems with writing? → when is buffer available for re-use?

Buffering — single

principle: user process can process one data block while next block is read in

swapping: can occur since input is taking place in system memory, not user memory

stream-oriented: buffer = input line, carriage return signals end of line

block-oriented:

- input transfers made to *system buffer*
- buffer moved to *user space* when needed
- another block read into system buffer

Buffering — double

principle: use 2 system buffers instead of 1 (per user process)

user process can write/read from one buffer while OS empties/fills other buffer

Buffering — circular

problem: double buffer insufficient for high-burst traffic situations:

- many writes between long periods of computations
- long computation periods while receiving data
- might want to read ahead more than just single block from disk

OS Structures

Monolithic Systems

advantages:

- well understood
- easy access to all system data (all shared)
- low module interaction cost (procedure call)
- extensible via interface definitions

disadvantages:

- no protection between system and application
- not stable/robust

Layered Systems

principle: system is divided into many *layers*:

- *each layer* uses functions and services of lower levels
- *bottom layer* = hardware
- *top layer* = user interface
- *lower layers*: implement mechanisms
- *higher layers*: implement policies (mostly)

advantages:

- *modular*: each layer can be tested/verifies independently
- *correctness* of layer n only depends on layer $n - 1$ → simple debugging/maintenance

disadvantages:

- just unidirectional protection
- mutual dependencies prevent strict layering

Monolithic Kernels

advantages:

— well understood
— performance OK
— sufficient protection between applications
— extensible via definitions + static/loadable modules

disadvantages:

— no protection between kernel components
— side-effects by undocumented interfaces
— complexity due to high degree of interdependency

Micro-Kernels

advantages:

— easier to test/prove/modify
— improved robustness/security
— improved maintainability
— coexistence of several APIs
— natural extensibility

disadvantages:

— additional decomposing
— low performance due to communication overhead

Virtual Machines

principle: takes layered approach to logical conclusion — treats hardware + OS kernel as like they were hardware

VM provides *identical* interface to underlying bare hardware

OS host creates illusion that process has own processor, memory,…

each guest gets (virtual) copy of underlying computer

benefits:

— multiple execution environments can share same hardware
— protection
— controllable file sharing
— use networking to communicate with each other
— useful for development/testing