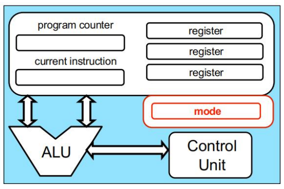
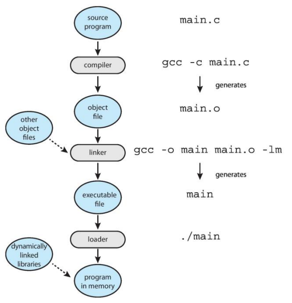
Von Neumann architecture model： A processor that performs operations and controls all that happens； A memory that contains code and data； I/O of some kind

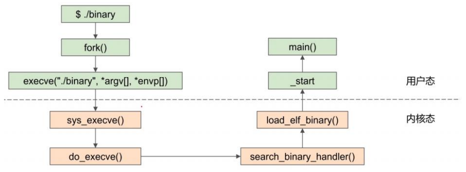


What is an OS： resource abstractor and a resource allocator(The OS defines a set of logical resources that correspond to hardware resources, and a set of well-defined operations on logical resources;OS decides who gets what resource and when)。
How does one start an OS：run a first program: the bootstrap program （ Stored in Read Only Memory (ROM)；运行流程：1. initializes the computer Register content, device controller contents 等。2. locates and loads the OS kernel into memory。3.The kernel starts the first process。4. nothing happens until an event occurs）
Multi-Programming: allow multiple“jobs”(running programs) to reside in memory simultaneously (context-switch: The OS picks and begins to execute one of the jobs in memory;When the job has to wait for “something”, then the OS picks another job to run)。
Time-Sharing: Multi-programming with rapid context-switching(Allows for interactivity:Response time very short;Each job has the illusion that it is alone on the system)。
A process is a running program(A process is a program in execution)。Running the OS Code?： The kernel is NOT a running job； It’s code that resides in memory and is ready to be executed at any moment When some event occurs； It can be executed on behalf of a job whenever requested； It can do special/dangerous things having to do with hardware
How OS achieves： protected (or privileged) instructions(only the OS can:Directly access I/O devices ;Manipulate memory management state;Manipulate protected control registers;Execute the halt instruction that shuts down the processor)
User vs. Kernel Mode: Unprivileged mode: protected instructions cannot be executed; Privileged mode: In this mode all instructions can be executed
OS Events: an “unusual” change in control flow。An event stops execution, changes mode, and changes context(it starts running kernel code); The kernel defines a handler for each event type(a piece of code executed in kernel mode); Once the system is booted, all entries to the kernel occur as the result of an event; There are two kinds of events: interrupts and exceptions (traps)。 Interrupts are caused by external events(Hardware-generated)； Exceptions are caused by executing instructions(Software-generated interrupts)
System Calls: When a user program needs to do something privileged。A system call is a special kind of trap。过程：Causes a trap, which maps to a kernel handler；Passes a parameter determining which system call to use；Saves caller state (PC, regs, mode) so it can be restored later
Timers： interrupts the computer regularly to make sure that an interrupt will occur reasonably soon(interrupts the computer regularly;Modifying the timer is done via privileged instructions)
Main OS Services: Process Management(Creating and deleting processes;Suspending and resuming processes;Providing mechanisms for process synchronization;Providing mechanisms for process communication;Providing mechanisms for deadlock handling);Memory Management(determines what is in memory when;Keeping track of which parts of memory are currently being used and by which process;Deciding which processes and data to move into and out of memory;Allocating and deallocating memory space as needed;The OS is not responsible for memory caching, cache coherency which managed by the hardware);Storage Management(operates File-System management:Creating and deleting files and directories; Manipulating files and directories; Mapping files onto secondary storage; Backup files onto stable (non-volatile) storage media; Free-space management; Storage allocation; Disk scheduling);I/O Management(hides peculiarities of hardware devices from the user;);Protection and Security(Memory protection, device protection;User IDs associated to processes and files;Group IDs for sets of users;Definition of privilege levels)
which of these instructions should be privileged, and why?: Set value of the system timer;Read the clock;Clear memory;Issue a system call instruction;Turn off interrupts;Modify entries in device-status table;Access I/O device
OS Services and Features: Helpful to users: program execution(Load and run a program; Allow a program to end in multiple ways e.g.with error codes);I/O operations(Allow programs access to I/O devices);files system(Provides file/directory abstractions; Allow programs to create/delete/read/write; Implements permissions); communacation(Provides abstractions for processes to exchange information); error detection;resource allocation(Decides which process gets which resource when); accounting(Keeps track of how much is used by each user); protection and security(Controls access to resources; Enforces authentication of all users; Allows users to protect their content) (绿色表示 Better efficiency/operation；蓝色表示 Helpful to users)
User Operating System Interface - CLI or command interpreter:allows direct command entry。GUI： User-friendly desktop metaphor interface
System Calls write() system call go through： syscall instruction on x86_64→Kernel space(kernel_entry Saved all user space registers; calls write syscall handler from syscall_table)→After write finish, call ret_to_user(Restore all saved user space registers;Transfer control flow to user space)
System Call Implementation:System-call interface maintains a table indexed according to system call numbers ;The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values;The caller need know nothing about how the system call is implemented(Just needs to obey API and understand what OS will do as a result call;Most details of OS interface hidden from programmer by API)
Time Spent in System Calls. three times: “real” time: wall-clock time (also called elapsed time, execution time, run time, etc.); “user” time: time spent in user code (user mode); “system” time: time spent in system calls (kernel mode)
System Call Parameter Passing. Three general methods:pass the parameters in registers;Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register;Parameters placed onto the stack by the program and popped off the stack by the operating system(Block and stack methods do not limit the number or length of parameters being passed)
Types of System Calls:File management(create file, delete file;open, close file;read, write, reposition;get and set file attributes);Device management(request device, release device;read, write, reposition;get device attributes, set device attributes;logically attach or detach devices);Information maintenance(get time or date, set time or date;get system data, set system data;get and set process, file, or device attributes);Communications(create, delete communication connection;send, receive messages if message passing model to host ;name or process name;Shared-memory model create and gain access to memory regions;transfer status information;attach and detach remote devices);Protection(Control access to resources;Get and set permissions;Allow and deny user access)
Linkers and Loaders. Linker combines relocatable object file into single binary executable file also brings in libraries(Modern systems dynamically linked libraries in Windows, DLLs are loaded as needed
Program resides on secondary storage as binary executable must be brought into memory by loader to be executed(Relocation assigns final addresses to program parts and adjusts code and data in program to match those addresses)

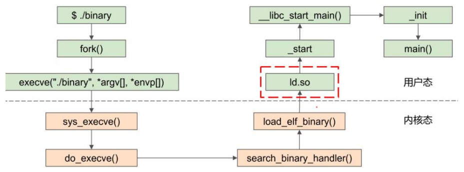


ELF binary basics:Executable and Linkable Format. .text: code; .rodata: initialized read-only data; .data: initialized data; .bss: uninitialized data
Where does static variable go? so: .data ; Static const? so: .rodata
Linking: Static linking(All needed code is packed in single binary, leading to large binary);Dynamic linking(Reuse libraries to reduce ELF file size) How to resolve library calls? It is the loader who resolves lib calls.(Statically-linked ELF has no .interp section Dynamically-linked ELF has .interp section). Who setups ELF file mapping?Kernel/exec syscall
Who setups stack and heap? Kernel/exec syscall Who setups libraries?Loader

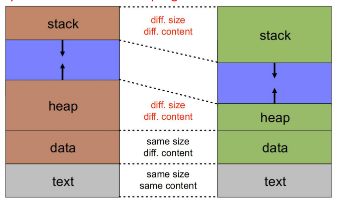
Running statically-linked ELF:



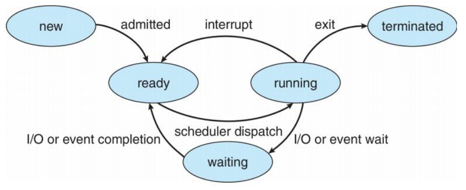
Running dynamically-linked ELF:



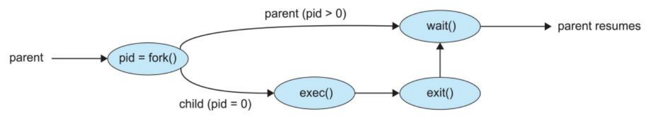
Static link vs dynamic link. Running statically-linked ELF:All code are packed in single binary, leading to large binary;Does not require dynamic loader;_start is executed after evecve system call.   Running dynamically-linked ELF : Reuse libraries to reduce ELF file size
Why Applications are Operating System Specific:Apps compiled on one system usually not executable on other operating systems(Each operating system provides its own unique system calls)
Application Binary Interface (ABI) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc
Operating System Design and Implementation:User goals: operating system should be convenient to use, easy to learn, reliable, safe, and fast; System goals :operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient. The separation of policy from mechanism :A very important principle; Allows policy changes without changed implemented
mechanism
Operating System Structure. Monolithic Structure(Unix, Linux:consists of two separable parts, Systems programs and the kernel consists of everything below the system-call interface and above the physical hardware provides the file system, CPU scheduling, memory management, and other operating-system functions),Monolithic supports loadable kernel module;Layered(The operating system is divided
into a number of layers , each built on top of lower layers. The bottom layer is the hardware; the highest is the user interface. With modularity, layers are selected such that each uses functions and services of only lower-level layers); Microkernels(Moves as much from the kernel into user space; Communication takes place between user modules using message passing). Benefits: Easier to extend a microkernel
; Easier to port the operating system to new architectures; More reliable (less code is running in kernel mode); More secure. Detriments:Performance overhead of user space to kernel space communication
Modules:Many modern operating systems implement loadable kernel modules (LKMs)(Uses object-oriented approach; Each core component is separate; Each talks to the others over known interfaces; Each is loadable as needed within the kernel)
Process： a unit of resource allocation and protection; A process is a program in execution; A process is a unit of resource allocation and protection(program is passive entity andbecomes a process when it’s loaded in memory). Process = code(initially stored on disk in an executable file) + data section(global variables (.bss and .data in x86 assembly)) + program counter + content of the processor’s registers
+ a stack + a heap(for dynamically allocated memory)
The Stack:The runtime stack is a stack on which items can be pushed or popped; The items are called activation records(stack frames);The stack is how we manage to have programs place successive function/method calls;The management of the stack is done entirely on your behalf by the compiler
2 processes for the same program:



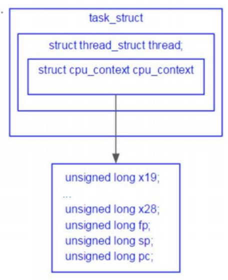
Process Control Block (PCB):Information associated with each process;Each process has and only has a PCB;包括：Process state(running, waiting)、Program counter、CPU registers、CPU scheduling information(priorities, scheduling queue pointers)、Memory-management information(memory allocated to the process)、Accounting information(CPU used, clock time elapsed since start, time limits)、I/O status information(I/O devices allocated to process, list of open files)
Process State： New: The process is being created； Running: Instructions are being executed； Waiting: The process is waiting for some events to occur； Ready: The process is waiting to be assigned to a processor；Terminated: The process has finished execution
Process Creation： Each process has a pid; ppid refers to the parent’s pid
fork():creates a new process; child is is a copy of the parent but It has a different pid (and thus ppid);Its resource utilization (so far) is set to 0; fork() returns the child’s pid to the parent, and 0 to the child

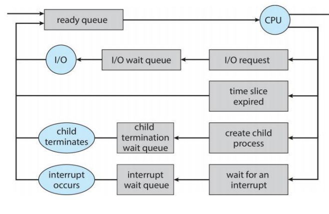


execve() system call used after a fork() to replace the process memory space with a new program
Parent process calls wait() for the child to terminate

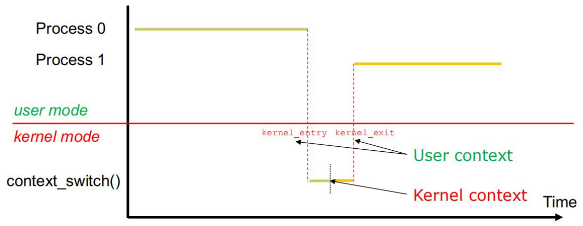


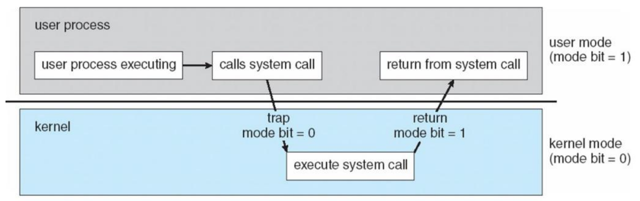
fork()优点：简洁(Windows CreateProcess 需提供 10 个参数);分工(fork 搭起骨架，exec 赋予灵魂);联系：保持进程与进程之间的关系
缺点：复杂(两个系统调用)；性能差；安全性问题
Process Terminations:用 exit() system call(This call takes as argument an integer that is called the process’exit/return/error code)、All resources of a process are deallocated by the OS、A process can cause the termination of another process
wait()： The wait() call blocks until any child completes ,returns the pid of the completed child and the child’s exit code. waitpid() :blocks until a specific child completes,can be made non-blocking with WNOHANG options
Processes and Signals:asynchronous event that the program must act upon, in some way
Zombie:When a child process terminates remains as a zombie in an “undead” state until it is “reaped” (garbage collected) by the OS. The OS keeps zombies around for this purpose:They’re not really processes, they do not consume resources; They only consume a slot in memory which may eventually fill up and cause fork() to fail Getting rid of zombies:The parent associates a handler to SIGCHLD; The handler calls wait()
Orphans:An orphan process is one whose parent has died,then the orphan is “adopted” by the process with pid 1,The process with pid 1 does handle child termination with a handler for SIGCHLD that calls wait (如果我们想创建一个守护进程 (daemon，在后台运行的、生存期长的进程，例如 host 一项服务等)，我们可以 fork 两次，让 grandchild 执行对应任务，而 child 直接终止，这样 grandchild 就会成为孤儿从而被 init 收养。)
Question: what resources cannot be deallocated by the child process? so: PCB
Process Scheduling:Process scheduler selects among ready processes for next execution on CPU core



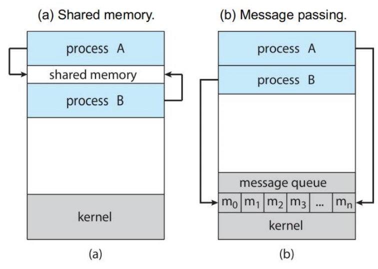


Context Switch:When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch(The more complex the OS and the PCB → the longer the context switch； Some hardware provides multiple sets of registers per CPU →multiple contexts saved/loaded at once)
Where does cpu_switch_to() return to? When the value is set? so： Return to the caller of cpu_switch_to à eventually to schedule()
When and where are the kernel context (regs) been saved?
When: In context_switch, more specifically, in cpu_switch_to
Where: In PCB, more specifically, in cpu_context
When and where are the user context (regs) been saved?
When: kernel_entry; Where: per-thread kernel stack, more specifically pt_regs

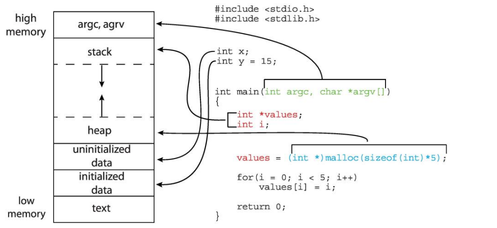




fork() return values. How does fork() return two values? For parent process, fork is just a syscall, similar to write:new_pid is returned to parent via syscall return value (saved in pt_regs);For child process, how does fork() return zero?Also via pt_regs, pt_regs[0] = 0; set the return value to 0.
When does child process start to run and from where? When forked, child is READY → context switch to RUN,After context switch, run from ret_from_fork -> ret_to_user -> kernel_exit who restores the pt_regs
Linux 2.3.0 (1999)(task_struct definition;schedule();switch_to() definition;Fixed PCB table, max 512;unnamed ready queue;named wait queue)Linux 2.4.0(task_struct number can be dynamic;Has the named ready queue Called runqueue_head;Links task_struct together, using task_struct->run_list) Linux 2.6.0(Go through priority array to pick the next)

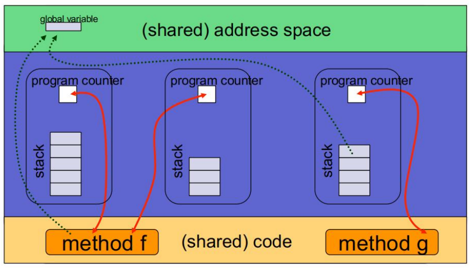


Inter-process Communication(IPC):The means of communication for cooperating processes. Reasons for cooperating processes:Information sharing;Computation speedup;Modularity;Convenience
IPC Communication Models. Shared Memory 特点： low-overhead(a few syscalls initially, and then none);more convenient for the user since we're used to simply reading/writing from/to RAM;more difficult to implement in the OS. Shared Memory 实现： Processes need to establish a shared memory region(contrary to the memory protection idea)→Processes communicate by reading/writing to the shared memory region. Message-passing 特点： useful for exchanging small amounts of data; simple to implement in the OS; sometimes cumbersome for the user as code is sprinkled with send/recv operations; high-overhead: one syscall per communication operation. Message-passing 实现过程： establish a communication "link" between processes→place calls to send() and recv()→optionally shutdown the communication "link" .Message-passing 实现方式. Physical:Shared memory;Hardware bus; Network/ Logical:Direct or indirect;Synchronous or asynchronous;Automatic or explicit buffering
Direct Communication Message Passing: Processes must name each other explicitly(send (P, message));Properties of communication link(Links are established automatically/A link is associated with exactly one pair of communicating processes/Between each pair there exists exactly one link/The link may be unidirectional, but is usually bi-directional). Indirect Communication Message Passing: Messages are directed and received from mailboxes(Each mailbox has a unique id/Processes can communicate only if they share a mailbox); Properties of communication link(Link established only if processes share a common mailbox/A link may be associated with many processes/Each pair of processes may share several communication links/Link may be unidirectional or bi-directional) Mailbox sharing 问题：P1, P2, and P3 share mailbox A，P1, sends; P2 and P3 receive，Who gets the message? Solutions：Allow a link to be associated with at most two processes；Allow only one process at a time to execute a receive operation； Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was. . Message passing is key for distributed computing
Synchronization. Blocking is considered synchronous(Blocking send -- the sender is blocked until the message is received/Blocking receive -- the receiver is blocked until a message is available);
Non-blocking is considered asynchronous(Non-blocking send -- the sender sends the message and continue / Non-blocking receive -- the receiver receives valid message, or Null message)
Buffering:Queue of messages attached to the link. three ways:Zero capacity(no messages are queued on a link.Sender must wait for receiver); Bounded capacity(finite length of n messages. Sender must wait if link full); Unbounded capacity(infinite length Sender never waits)
Signals:used to notify a process that some events has occurred Pipes:Acts as a conduit allowing two processes to communicate. Ordinary pipes:cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created. Named pipe:can be accessed without a parent-child relationship.
Client-Server Communication:Applications are often structured as sets of communication processes
Sockets: A socket is a communication abstraction with two endpoints so that two processes can communicate(Socket = ip address + port number)
Remote Procedure Calls:provides a procedure invocation abstraction across hosts(A "client" invokes a procedure on a "server" , just as it invokes a local procedure). It is done by a client stub, 步骤:marshals arguments→sends them over to a server→wait for the answer→wait for the answer what happens if the RPC fails？ Weak (easy to implement) semantic: at most once(Server maintains a time-stamp of incoming messages/If a repeated message shows up, ignore it); Strong (harder to implement) semantic: exactly once(The server must send an ack to the client/The client periodically retries until the ack is received)
Java RMI: Objects are serialized and deserialized; RMI sends copies of local objects and references to remote objects; RMI hides most of the gory details of IPCs



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

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Why thread? make a process run faster Thread Definition： A thread is a basic unit of execution within a process. Each thread has its own thread ID, program counter,register set ,Stack; It shares the following with other threads code section,data section,the heap (dynamically allocated memory),open files and signals
Advantages of Threads:Economy(Creating a thread is cheap,Much cheaper than creating a process,Code, data and heap are already in memory;Context switch between threads is cheap,No cache flush); Resource Sharing(Having concurrent activities in the same address space); Responsiveness(While one thread blocks waiting for
some event, another can do something); Scalability(Running multiple "threads" at once uses the machine more effectively)
Drawbacks of Threads:Weak isolation between threads: If one thread fails (e.g., a segfault), then the process fails; This leads to process-based concurrency; Threads may be more memory-constrained than processes; Threads do not benefit from memory protection(Concurrent programming with threads is hard)
Multi-Threading Challenges:Deal with data dependency and synchronization; Dividing activities among threads; Balancing load among threads; Split data among threads; Testing and debugging



Many-to-One Model. Advantage: multi-threading is efficient and low-overhead Drawback #1: cannot take advantage of a multi-core architecture #2: if one threads blocks, then all the others do
One-to-One Model. Not as fast as in the Many-to-One; Removes both drawbacks of the Many-to-One Model
Many-to-Many Model. If a user thread blocks, the kernel can create a new kernel threads to avoid blocking all user threads; A new user thread doesn't necessarily require the creation of a new kernel thread
Pthreads. provided either as user-level or kernel-level,API specifies behavior of the thread library, implementation is up to development of the library, Common in UNIX operating systems
Thread Scheduling: Question: who decides which runnable thread to run? Old versions of the JVM used Green Threads(Green threads have all the disadvantages of user-level threads , Cannot exploit multi-core, multi-processor architectures) The JVM now provides native threads
What happens when a thread calls fork()? A new process is created that has only one thread (the copy of the thread that called fork(),linux use), or A new process is created with all threads of the original process (a copy of all the threads, including the one that called fork())
Safe Thread Cancellation:Asynchronous cancellation(One thread terminates another immediately); Deferred cancellation(A thread periodically checks whether it should terminate)
Linux Threads: a thread is also called a light-weight process (LWP); The clone() syscall is used to create a thread or a process; Linux uses the same task_struct for process and thread(A process is a single thread + an address space or multiple threads + an address space(PID is the leading thread ID))

CPU Scheduling Definition:The decisions made by the OS to figure out which ready processes/threads should run and for how long 目的： Maximizes CPU utilization so that it' s never idle
CPU-I/O Burst Cycle： Most processes alternate between CPU and I/O activities(I/O-bound process: Mostly waiting for I/O,Many short CPU bursts; CPU-bound process Mostly using the CPU,Very short I/O bursts if any), processes are diverse makes CPU scheduling difficult
Scheduling Decision Points: #1:A process goes from RUNNING to WAITING; #2: A process goes from RUNNING to READY; #3: A process goes from WAITING to READY; #4: A process goes from RUNNING to TERMINATED #5: A process goes from NEW to READY; #6: A process goes from READY to WAITING. Non-preemptive scheduling: Except #2; Preemptive scheduling: #1, #2, #3, #4, #5
Scheduling Objectives:Maximize CPU Utilization; Maximize Throughput; Minimize Turnaround Time; Minimize Waiting Time; Minimize Response Time
Waiting time = start time – arrival time Turnaround time = finish time arrival time
First-Come, First-Served (FCFS): Convoy effect - short process behind long process,Long jobs slow down the whole system
Shortest-Job-First (SJF): SJF is optimal – gives minimum average waiting time for a given set of processes. Predicting CPU burst durations :

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0$$

Round-Robin: is preemptive and designed for time-sharing; It defines a time quantum,Unless a process is the only READY process, it never runs for longer than a time quantum before giving control to another ready process(average waiting time worse than SJF, but better response time,No starvation)
Priority Scheduling. SJF is priority scheduling where priority is predicted next CPU burst time + Round-Robin:Run the process with the highest priority. Processes with the same priority run round-robin
The problem: will a low-priority process ever run A solution: Priority aging (Increase the priority of a process as it ages)
Multilevel Queue Scheduling:Processes can move among the queues(Number of queues; Scheduling algorithm for each queue; Scheduling algorithm across queues; Method used to promote/demote a process) Multilevel Feedback Queues
Multiple-Processor Scheduling – Load Balancing:Load balancing attempts to keep workload evenly distributed; Push migration:periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs; Pull migration idle processors pulls waiting task from busy processor
Linux Scheduling:Linux 0.11(Round-robin + priority scheduling;Implemented with an array;TASK_RUNNING means ready) Linux 1.2(Implemented with a circular queue) Linux 2.2(Scheduling classes,real-time, non-preemptible, non-real-time;Priorities within classes) Linux Scheduling: 2.4(The schedule proceeds as a sequence of epochs; How to compute time slices?If a process uses its whole time slice, then it will get the same one,If a process hasn' t used its whole time slice (.e.g., because blocked on I/O) then it gets a larger time slice!;Problem: At each scheduling event, the scheduler needs to go through the whole list of ready tasks to pick one to run/If n (the number of tasks) is large, then it will take long to pick one to run) Linux 2.6(O(1) scheduler:Using a Bitmap for Speed,Finding the highest priority for which there is a ready task becomes simple,just find the first bit set to 1 in the bitmap; active task: its time slice hasn' t been fully consumed/expired task: has used all of its time slice)
Linux ≥ 2.6.23(keep track of how fairly the CPU has been allocated to tasks, and "fix" the unfairness;For each task, the kernel keeps track of its virtual time;Tasks are stored in a red-black tree, O(log n) time to retrieve the least happy task,O(1) to update its virtual time once it' s done running for a while,O(log n) time to re-insert it into the red-black tree)

Solution to Critical-Section. Three Requirements : Mutual Exclusion(only one process can execute in the critical section); Progress(if no process is executing in its critical section and some processes wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely); Bounded waiting(There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted)
Peterson's Solution:solves two-processes/threads synchronization; It assumes that LOAD and STORE are atomic; Two processes share two variables boolean flag[2]: whether a process is ready to enter the critical section int turn: whose turn it is to enter the critical section

- P0:

```
do {
    flag[0] = TRUE;
    turn = 1;
    while (flag[1] && turn == 1);
    critical section
    flag[0] = FALSE;
    remainder section
} while (TRUE);
```

- P1:

```
do {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] && turn == 0);
    critical section
    flag[1] = FALSE;
    remainder section
} while (TRUE);
```

Memory Barrier:重排可能使得在多核运行时出现与期望不同的结果。为了解决这个问题，Memory Barrier 保证其之前的内存访问先于其后的完成。即，我们保证在此前对内存的改变对其他处理器上的进程是可见的。
Test-and-Set Instruction: 如果 lock 在 Entry Section 时为 true，那么 test_and_set(&lock) 将返回 true，因此会始终在 while 循环中询问。直到某个时刻 lock 为 false，那么 test_and_set(&lock) 将返回 false 同时将 lock 置为 true，进程进入 Critical Section，同时保证其他进程无法进入 Critical Section。当持锁的进程完成 Critical Section 的运行，它在 Exit Section 中释放

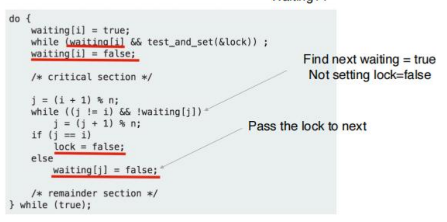
```
bool test_set (bool *target)
{
    bool rv = *target;
    *target = TRUE;
    return rv:
}
```

```
bool lock = FALSE
do {
    while (test_set(&lock));   // busy wait
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

lock

Test-and-Set Lock 可以作如下更改以满足 bounded waiting：引入了 bool 数组 waiting[] 。在 Entry Section 中，我们首先置 waiting[i] 为 true；当 waiting[i] 或者 lock 中任意一个被释放时，进程可以进入 Critical Section。初始时， lock 为 false，第一个请求进入 CS 的进程可以获许运行。在 Exit Section 中，进程从下一个进程开始，遍历一遍所有进程，发现正在等待的进程时释放它的 waiting[j] ，使其获许进入 CS，当前进程继续 Remainder Section 的运行；如果没有任何进程在等待，那么它释放 lock ，使得之后第一个请求进入 CS 的进程可以直接获许。这样的方式可以保证每一个进程至多等待 n-1 个进程在其前面进入 CS

WaitingT1



```
do {
    waiting[i] = true;
    while (waiting[i] && test_and_set(&lock)) ;
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */
} while (true);
```

Compare-and-Swap Instruction:Shared integer lock initialized to 0;

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;

    if (*value = expected)
        *value = new_value;
    return temp;
}

 while (true)
 {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;
    /* remainder section */
 }
```

Atomic Variables: provides atomic (uninterruptible) updates on basic data types such as integers and booleans

```
void increment(atomic_int *v) {
    int temp;

    do {
        temp = *v;
    } while (temp != (compare_and_swap(v,temp,temp+1));
}
```

Mutex Locks:

```
bool locked = false;

acquire() {
    while (compare_and_swap(&locked, false, true))
        ; //busy waiting
}

release() {
    locked = false;
}
```

Mutex Locks busy waiting：T0 acquires lock -> INTERRUPT-> T1 runs, spin, spin spin … (till time's out) -> INTERRUPT-> T0 runs -> INTERRUPT->T1 runs, spin, spin spin … -> INTERRUPT-> T0 runs, release locks -> INTERRUPT -> T1 runs, enters CS

sulotion:yield-> moving from running to sleeping(如果所需的等待时间一般小于 context switch 所需的时间的话，用 spinlock 可能是更好的)

Semaphore:

Definition of the **wait() operation**

```
wait(S) {
    while (S <= 0) ; // busy wait
    S--;
}
```

Definition of the **signal() operation**

```
signal(S) {
    S++;
}
```

Implementation with waiting queue:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

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

Comparison between mutex and semaphore Mutex Pros: no blocking; Cons: Waste CPU on looping Good for short critical section Semaphore Pros: no looping Cons: context switch is time-consuming Good for long critical section How to achievethe atomic?•Spinlock

Priority Inversion: three processes, PL, PM, and PH with priority PL < PM < PH• PL holds a lock that was requested by PH ⇨ PH is blocked• PM becomes ready and preempted the PL• It effectively "inverts" the relative priorities of PM and PH

Solution: priority inheritance(如上例中，priority inheritance 将允许 L 临时继承 H 的优先级从而防止被 M 抢占；当 L 释放锁后则回到原来的优先级，此时 H 将在 M 之前执行。)

Linux Synchronization:prior to version 2.6, disables interrupts to implement short critical sections; version 2.6 and later, fully preemptive; Linux provides:• atomic integers• spinlocks• semaphores• reader-writer locks(read-copy-update (RCU) is a synchronization mechanism based on mutual exclusion. It is used when performance of reads is crucial and is an example of space – time tradeoff, enabling fast operations at the cost of more space.)

Bounded-Buffer Problem:

```
semaphore lock = 1;
semaphore eslot = BUFFER_SIZE;
semaphore fslot = 0;

producer() {
    while (true) {
        wait(eslot);    // if buffer is full, i.e. eslot == 0, wait
                        // else, eslot--
        wait(lock);
        add_to_buffer(next_produced);
        signal(lock);
        signal(fslot);  // fslot++
    }
}

consumer() {
    while (true) {
        wait(fslot);    // if buffer is empty, i.e. fslot == 0, wait
                        // else, fslot--
        wait(lock);
        next_consumed = take_from_buffer();
        signal(lock);
        signal(eslot);  // eslot++
    }
}
```

Readers-Writers Problem:

```
semaphore write_lock = 1;
int reader_count = 0;
semaphore reader_count_lock = 1;
semaphore writer_first = 1;

writer() {
    while (true) {
        wait(writer_first);
        wait(write_lock);
        read_and_write();
        signal(write_lock);
        signal(writer_first);
    }
}

reader() {
    while (true) {
        wait(writer_first);
        wait(reader_count_lock);
        reader_count++;
        if (reader_count == 1)
            wait(write_lock);
        signal(reader_count_lock);
        signal(writer_first);

        read();

        wait(reader_count_lock);
        reader_count--;
        if (reader_count == 0)
            signal(write_lock);
        signal(reader_count_lock);
    }
}
```

Deadlock：a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

Four Conditions of Deadloc. Mutual exclusion: a resource can only be used by one process at a time Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes No preemption: a resource can be released only voluntarily by the process holding it, after it has completed its task

Circular wait: there exists a set of waiting processes {P0, P1, …, Pn}

Basic Facts:• If graph contains no cycles ➡ no deadlock• If graph contains a cycle • if only one instance per resource type, ➡ deadlock • if several instances per resource type ➡ possibility of deadlock

Deadlock Prevention. prevent mutual exclusion:• not required for sharable resources• must hold for non-sharable resources; prevent hold and wait: whenever a process requests a resource, it doesn’thold any other resources(require process to request all its resources before it begins execution; allow process to request resources only when the process has none)➡ low resource utilization; starvation possible handle no preemption(if a process requests a resource not available➡ release all resources currently being held,preempted resources are added to the list of resources it waits for, process will be restarted only when it can get all waiting resources) handle circular wait(impose a total ordering of all resource types→require that each process requests resources in an increasing order)

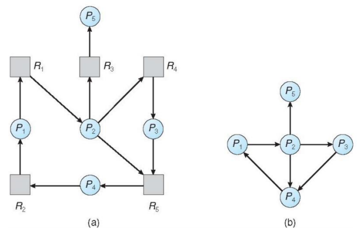
Deadlock Avoidance. 需要一些额外信息，例如进程未来需要使用哪些资源、资源的使用顺序等 ，资源分配状态 (resource allocation state):每个进程声明可能对每种资源类型的 最大需求 (maximum demands)/当前系统的 available 和 allocated 的资源数目。

Safe State.如果系统能够按照一定顺序为每个进程分配资源，同时避免死锁，那么系统就处在安全状态 (safe state)。即：there exists a sequence <P1, P2, …, Pn> of all processes in the system，for each Pi, resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj

Deadlock Avoidance Algorithms• Single instance of each resource type ➡ use resource-allocation graph • Multiple instances of a resource type ➡ use the banker’s algorithm

Deadlock Detection.:Allow system to enter deadlock state, but detect and recover from it Single Instance Resources:在图里找环:

Multi-instance. :银行家算法



Deadlock Recovery. 选择 1：Terminate deadlocked processes.•放弃所有死锁进程。这样的花费会很大•每次放弃一个进程，直到死锁环解除。这样的花费也很大，因为每次放弃一个进程之后需要调用死锁检测算法。参考如下指标选择造成的代价最小的进程来终止：进程的优先级；已经算了多久，还要算多久；用了哪些、多少资源，是否容易抢占；还需要多少资源；终止这一进程的话还需要终止多少进程；进程是交互的还是批处理的

选择 2：Resource preemption(不断抢占资源给其他进程用，直到消除死锁环为止)需要考虑三个问题：•选择牺牲进程 (Select a victim)。抢占哪些进程的哪些资源；•回滚(Rollback)。当一个进程的若干资源被抢占，我们需要将这个进程回滚到某个安全状态，即回滚到申请那些被抢占的资源之前。

一般来说，很难确定什么是安全状态，最简单的方案就是完全回滚，也就是终止进程并重新执行。回滚到足够打断死锁的状态更加经济，但是需要系统保存更多资源；•饥饿 (Starvation),如何保证不会永远从一个进程中抢占资源？在代价评价中增加回滚次数，也类似于 priority aging。

Partition. Base and Limit registers Hardware Address Protection:CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

Advantages: Built-in protection provided by Limit(No physical protection per page or block); Fast execution(Addition and limit check at hardware speeds within each instruction)

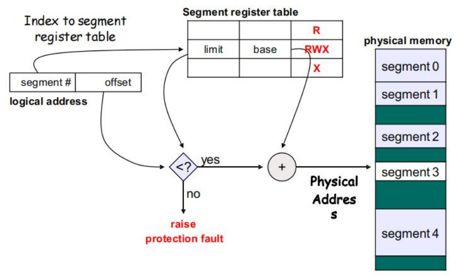
Fast context switch(Need only change base and limit registers); No relocation of program addresses at load time(All addresses relative to zero); Partition can be suspended and moved at any time(Process is unaware of change,Expensive for large processes)

Fixed partitions. divide memory into equal sized pieces (except for OS); Degree of multiprogramming = number of partitions; Problem – Internal Fragmentation

Variable partitions. Memory is dynamically divided into partitions based on process needs➡操作系统维护一个表，记录可用和已用的内存。最开始时整个内存就是一大块可用的内存块（将可用内块称为 hole）；经过一段时间的运行后，内存可能会包含一系列不同大小的孔. Problem – External Fragmentation

根据一组 hole 来分配大小为 n 的请求，称为 dynamic storage-allocation problem。这个问题最常用的解决方法包括：First-fit -分配首个足够大的 hole。这种方法会使得分配集中在低地址区，并在此处产生大量的碎片，在每次尝试分配的时候都会遍历到，增大查找的开销。best-fit -分配最小的足够大的 hole。除非空闲列表按大小排序，否则这种方法需要对整个列表进行遍历。这种方法同样会留下许多碎片。worst-fit -分配最大的 hole。同样，除非列表有序，否则我们需要遍历整个列表。这种方法的好处是每次分配后通常不会使剩下的空闲块太小，这在中小进程较多的情况下性能较好，并且产生碎片的几率更小。

Segmentation. 程序中的主函数、数组、符号表、子函数等等内部需要有一定的顺序，但是这些模块之间的先后顺序是无关紧要的,一个程序是由一组 segment （段）构成的，每个 segment 有其名称和长度。只要知道 segment 在物理内存中的基地址 (base) 和段内偏移地址 (offset) 就可以对应到物理地址中了。对于每一个 segment，我们给其一个编号。即，我们通过二元有序组 表示了一个地址。Logical address consists of a pair: <segment-number, offset>



Logical vs. Physical Address.•Logical address – generated by the CPU; also referred to as virtual address• Physical address – address seen by the memory unit

分段仍然存在 external fragmentation

解决问题：将内存重排使得 holes 连成一块；或者设计方案让程序不再需要连续的地址。

Compaction 将内存中的内容重排使得所有空闲空间连续。这一操作要求内存中的程序是 relocatable 的，即其地址是相对 base 的偏移；需要将内存逐一复制，这将消耗很多时间。

考虑「让程序不再需要连续的地址」。实际上，分段已经是这个方向上做出的一种尝试了，因为它将程序分为了几块，相比于简单的 partition，分段有助于减小 external fragmentation。

Paging: basic idea：Contiguous → Noncontiguous • Fixed and variable partitions are physical contiguous allocation • Avoids external fragmentation • Avoids problem of varying sized memory chunks

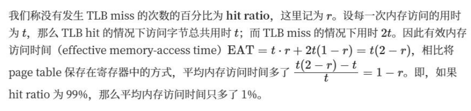
Basic Method 将 physical memory 切分成等大小的块，称为 frames；将 logical memory 切分成同样大小的块，称为 pages。当一个进程要执行时，其内容填到一些可用的 frame 中，其中的每一个地址可以用这个 frame 的 base 或 number（由于 frame 是被等大切分出来的，因此每个 frame 的 base 也唯一对应一个 frame number）以及相对这个 base 的 offset 表示；同时 CPU 生成逻辑地址，逻辑地址包含一个 page number 和一个 page offset；另有一个 page table，它以 page number 索引，其中的第 i 项存储的是 page number 为 i 的 page 所在物理内存的 frame 的 base。这样，每一个 page 将通过其 page number 映射到一个 frame 上；进而 page 中的每一个地址也通过 offset 与对应的 frame 建立映射。

Internal Fragmentation:Paging has no external fragmentation, but internal fragmentation (average internal fragmentation: 1 / 2 frame size • In practice, average is much smaller • Program contains several pages, only last page has fragmentation)

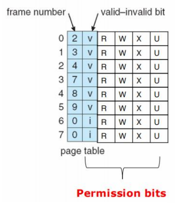
Small frame sizes vs large frame size? Small size means less internal fragmentation, but more page table entries

Page Table:Stores the logical page to physical frame mapping(• Which frame is free, and how many frames have been allocated • One entry for each physical frame • The allocated frame belongs to which process)

Page Table Hardware Support: Simplest Case. 一组专用的寄存器来实现,优点:使用时非常迅速(因为对寄存器的访问是十分高效的；缺点：由于成本等原因，寄存器的数量有限，因此这种方法要求 page table 的大小很小；由于专用寄存器只有一组，因此 context switch 时需要存储并重新加载这些寄存器。
Page table in memory & PTBR. 将页表放在内存中: Page-Table Base Register (PTBR) 指向页表;page-table length register (PTLR) indicates the size of the page table。在 context switch 时只需要修改 PTBR 问题:首先要根据 PTBR 和 page number 来找到页表在内存的位置，根据 frame number 和 page offset 访问对应的字节内容，访问需要两次内存访问，加倍内存访问的时间
解决方法: translation look-aside buffer (TLB) • if page number is in the TLB, no need to access the page table • if page number is not in the TLB, need to replace one TLB entry • TLB usually use a fast-lookup hardware cache called associative memory • TLB is usually small, 64 to 1024 entries TLB with ASID :每个 process 都有其自己的 page table,切换进程时也需要切换 page table(需要保证 TLB 与当前进程的 page table 是一致的)。选择 1：在每次切换时 flush TLB。选择 2：有些 TLB 在每个条目中保存 Address-Space Identifier (ASID)，每个 ASID 唯一标识一个进程。当 TLB 进行匹配时，除了 page number 外也对 ASID 进行匹配。
Associative Memory：memory that supports parallel search(Associative memory is not addressed by "addresses", but contents) if page# is in associative memory's key, return frame# (value) directly
Main TLB:• A 4-way, set-associative, 1024 entry cache which stores VA to PA mappings for 4KB, 16KB, and 64KB page sizes. • A 2-way, set-associative, 128 entry cache which stores VA to PA mappings for 1MB, 2MB, 16MB, 32MB, 512MB, and 1GB block sizes
TLB Match Process: • Its VA, moderated by the page size such as the VA bits[47:12], matches that of the requested address• The memory space matches the memory space state of the requests• The ASID matches the current ASID held in the CONTEXTIDR, TTBR0, or TTBR1 register or the entry is marked global• The VMID matches the current VMID held in the VTTBR register
Effective Access Time:

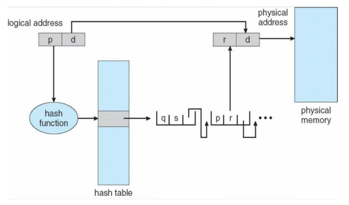

我们称没有发生 TLB miss 的次数的百分比为 hit ratio，这里记为 r。设每一次内存访问的用时为 t，那么 TLB hit 的情况下访问字节总共用时 t；而 TLB miss 的情况下用时 $2t$。因此有效内存访问时间（effective memory-access time）$\mathrm{EAT} = t \cdot r + 2t(1-r) = t(2-r)$，相比将 page table 保存在寄存器中的方式，平均内存访问时间多了 $\frac{t(2-r)-t}{t} = 1-r$。即，如果 hit ratio 为 99%，那么平均内存访问时间只多了 1%。

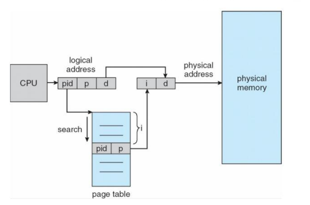
Memory Protection:Accomplished by protection bits with each frame; • Each page table entry has a present (aka. valid) bit (present: the page has a valid physical frame, thus can be accessed) • Each page table entry contains some protection bits(kernel/user, read/write, execution?, kernelexecution?) • Any violations of memory protection result in a trap to the kernel
Page Sharing:分页可以允许进程间共享代码，同一程序的多个进程可以使用同一份代码，只要这份代码是 reentrant code （or non-self-modifying code : never changes between execution）
Structure of Page Table. One-level page table can consume lots of memory for page table e.g. 32-bit logical address space and 4KB page size • page table would have 1 million entries (2^32 / 2^12)• if each entry is 4 bytes ➔ 4 MB of memory for page table alone
Two-Level Page Table: Example: 2-level paging in 32-bit Intel CPUs • 32-bit address space, 4KB page size• 10-bit page directory number, 10-bit page table number• each page table entry is 4 bytes, one frame contains 1024 entries (2^10)
64-bit Logical Address Space ARM64: 39 bits = 9+9+9+12



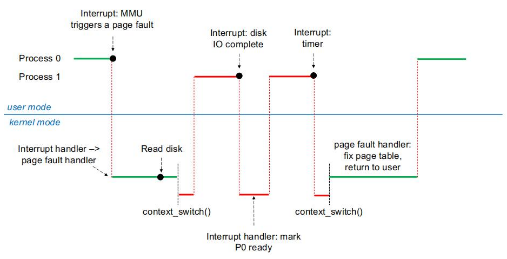
0	2	v	R	W	X	U
1	3	v	R	W	X	U
2	4	v	R	W	X	U
3	7	v	R	W	X	U
4	8	v	R	W	X	U
5	9	v	R	W	X	U
6	0	i	R	W	X	U
7	0	i	R	W	X	U

Hashed Page Tables. 哈希页表给 page#分配 frame#通过哈希计算得到。每个 page 对应的表项在内存中并不需要连续,哈希页表的每一个条目除了 page number 和 frame number 以外，还有一个指向有同一哈希值的下一个页表项的指针
Inverted Page Table. 整个系统只有一个页表，并且每个物理内存的 frame 只有一条相应的条目。寻址时，CPU 遍历页表，找到对应的 pid 和 page number，其在页表中所处的位置即为 frame number
Swapping. 将进程或进程的一部分暂时从内存 swap 到 backing store （备份存储，通常是一个比较快的磁盘）中，继续运行时从中重新拿回到内存。Swapping 使得所有进程总的物理地址空间总和超过系统中真实的物理地址空间称为可能，提高了系统的 degree of multiprogramming
问题：导致很长的 context switch 用时→只 swap out 一些 pages，而不必 swap out 一整个进程
Linux Support for Intel Pentium：Linux uses only 6 segments • kernel code, kernel data, user code, user data • task-state segment (TSS), default LDT segment; Uses a generic four-level paging for 32-bit and 64-bit systems • Two-level paging, middle and upper directories are omitted • Three-level page, upper directories are omitted

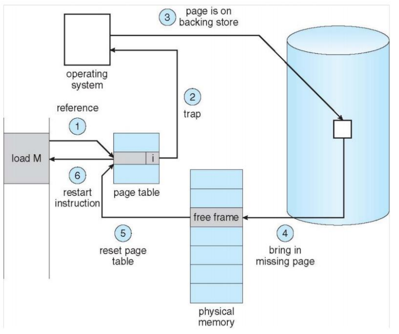




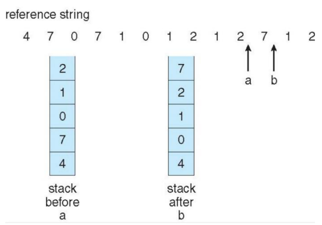
How about page size is 64KB • What is the virtual address format for 32-bit?(39 = 9+14+16) • What is the virtual address format for 64-bit?(48 = 4+13+13+16)
virtual Memory Background：Code needs to be in memory to execute, but entire program rarely needed or used at the same time • unused code: error handling code, unusual routines • unused data: large data structures→partially-loaded program(• program no longer constrained by limits of physical memory • programs could be larger than physical memory)
Virtual memory: separation of logical memory from physical memory(only part of the program needs to be in memory for execution • logical address space can be much larger than physical address space • more programs can run concurrently• less I/O needed to load or swap processes (part of it); allows memory (e.g., shared library) to be shared by several processes • better IPC performance; allows for more efficient process forking (copy-on-write))
Virtual-address Space: • Usually design virtual address space for stack to start at Max logical address and grow "down" while heap grows "up" • Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc • System libraries shared via mapping into virtual address space • Pages can be shared during fork(), speeding process creation: COW
Demand Paging. Demand paging brings a page into memory only when it is demanded(• demand means access (read/write) • if page is invalid (error) ➠ abort the operation • if page is valid but not in memory ➠ bring it to memory)
What causes page fault? • User space program accesses an address Which hardware issues page fault? • MMU Who handles page fault? • Operating system
MMU issues page fault: How does MMU know the physical frame is not mapped? Each page table entry has a valid–invalid (present) bit(• v ➠ frame mapped, i ➠ frame not mapped • initially, valid–invalid bit is set to i on all entries • during address translation, if the entry is invalid, it will trigger a page fault)
Page Fault swapper: Lazy swapper: never swaps a page in memory unless it will be needed; Pre-Paging: pre-page all or some of pages a process will need, before they are referenced(• it can reduce the number of page faults during execution • if pre-paged pages are unused, I/O and memory was wasted ,although it reduces page faults, total I/O# likely is higher)
Demand paging needs hardware support(• page table entries with valid / invalid bit • backing storage (usually disks) • instruction restart)
Stages in Demand Paging–Worse Case: 1. Trap to the operating system 2. Save the user registers and process state 3. Determine that the interrupt was a page fault 4. Check that the page reference was legal and determine the location of the page on the disk 5. Issue a read from the disk to a free frame(5.1 Wait in a queue for this device until the read request is serviced 5.2 Wait for the device seek and/or latency time 5.3 Begin the transfer of the page to a free frame) 6. While waiting, allocate the CPU to other process 7. Receive an interrupt from the disk I/O subsystem (I/O) completed 7.1 Determine that the interrupt was from the disk 7.2 Mark page fault process ready) 8. Handle page fault: wait for the CPU to be allocated to this process again(8.1 Save registers and process state for other process 8.2 Context switch to page fault process 8.3 Correct the page table and other tables to show page is now in memory)



Interrupt: MMU triggers a page fault
Interrupt: disk IO complete
Interrupt: timer
Process 0
Process 1
user mode
kernel mode
Interrupt handler –> page fault handler
Read disk
page fault handler: fix page table, return to user
context_switch()
context_switch()
Interrupt handler: mark P0 ready



9. Return to user: restore the user registers, process state, and new page table, and then resume the interrupted instruction
Demand Paging: EAT. Page fault rate: 0 ⩽ p ⩽ 1 Effective Access Time (EAT)= (1–p) x memory access + p x (page fault overhead + swap page out + swap page in + instruction restart overhead)
EX：Assume memory access time: 200 nanoseconds, average page-fault service time: 8 milliseconds
• EAT = (1 – p) x 200 + p x (8milliseconds) = (1 – p) x 200 + p x 8,000,000 = 200 + p x 7,999,800 → if one out of 1,000 causes a page fault, then EAT = 8.2 microseconds → a slowdown by a factor of 4100%–8.2 us / 0.2 us = 41 → if want < 10 percent, less than one page fault in every 400,000 accesses
Demand Paging Optimizations：Swap space I/O faster than file system I/O even if on the same device→Copy entire process image from disk to swap space at process load time; Demand page in from program binary on disk, but discard rather than paging out when freeing frame (and reload from disk next time) (Following cases still need to write to swap space • Pages not associated with a file (like stack and heap)–anonymous memory • Pages modified in memory but not yet written back to the file system)
Copy-on-Write:父进程和子进程最初使用同一份物理页来进行工作，在任何一个进程需要写入某个共享 frame 时再进行复制; Linux 等操作系统提供了 vfork()，进一步优化子进程在 fork() 之后立刻调用 exec() 的情形。vfork() 并不使用 copy-on-write；调用 vfork() 之后，父进程会被挂起，子进程使用父进程的地址空间。如果子进程此时修改地址空间中的任何页面，这些修改对父进程都是可见的。
What Happens if There is no Free Frame?交换出去一整个进程从而释放它的所有帧；找到一个当前不在使用的帧释放 基本步骤是：1.找到这个 victim frame；2.将其内容写到交换空间；2.修改页表（和 TLB 等）以表示它不在内存中了(now potentially 2 page I/O for one page fault ➠ increase EAT)
Page Replacement Algorithms. FIFO (First In First Out):换出最先进入内存的页面(使用一个队列保存调入内存的顺序即可) 问题:逻辑和实际不符,实际情况下有很多页面会经常被访问；物理帧增加的时候 page-fault 反而更多(Belady's Anomaly)
Optimal: 选择最长时间内不再被访问的页面换出(这种方案的 page-fault rate 是最低的。由于实际中没有办法预测结果，因此它只作为理论最优解用来判定其他算法的优劣)
LRU (Least Recently Used):换出最久没有被访问的页面(给每个页表项一个 counter，每次访问某个 page 时，将 counter 更新为当前的时间。每次需要置换时，搜索 counter 最小的页,可以用 heap 来优化;用一个栈保存 page numbers，每次访问时找到它然后把它挪到栈顶,这两种实现开销都比较大。)
LRU-Approximation:在 LRU 和性能之间折中；引入一个 reference bit，当一个 page 被访问时这个 bit 置 1；操作系统定期将 reference bit 清零。在需要交换时，找一个 reference bit 为 0 的说明它在这段时间内没有被访问过 EX:Suppose we have 8-bits byte for each page • During a time interval (100ms), shifts bit rights by 1 bit, sets the high bit if used, and then discards the low-order bits
• 00000000 => has not been used in 8 time intervals • 11111111 => has been used in all time intervals • 11000100 vs 01110111 : 01110111 is used more recently?
Second-chance algorithm: Generally FIFO, plus hardware-provided reference bit; If page to be replaced has • Reference bit = 0 -> replace it • reference bit = 1 then: – set reference bit 0, leave page in memory – replace next page, subject to same rules
Enhanced Second-Chance Algorithm. 引用位和修改位(引用位：表示页面是否最近被访问过。修改位：表示页面自上次被加载到内存以来是否被修改过)页面状态分类：(0, 0)：页面既没有最近被使用也没有被修改。(0, 1)：页面最近没有被使用，但已被修改。(1, 0)：页面最近被使用过但没有修改。(1, 1)：页面最近被使用过且已被修改。 替换过程：当需要替换页面时，算法根据上述四类状态在循环队列中寻找候选页面。(0, 0)→(0, 1)→(1, 0)→(1, 1)可能需要遍历循环队列多次，直到找到合适的替换页面。优点：不仅考虑了页面是否被访问过，还考虑了页面是否被修改。这使得算法能够更好地平衡内存中页面的使用频率和写回硬盘的成本。
Counting-based Page Replacement. Least Frequently Used (LFU):LFU 算法保持跟踪每个页面被引用的次数。当需要替换页面时，该算法选择引用次数最少的页面; 缺点：初始使用问题：如果一个页面在进程初始化期间被大量使用，它的计数器会很高。即使之后再也不使用，由于初始的高计数，它可能不会被替换/计数器溢出：长时间运行的系统可能导致计数器值非常高，导致计数溢出/频率动态变化：LFU 可能不会很好地反映最近的使用模式，因为它侧重于历史总体频率。Most Frequently Used (MFU):MFU 替换引用次数最多的页面，意义是引用次数最少的页面可能是最近才加载到内存中的;缺点：反直觉：在很多情况下，最常用的页面是最不应该被替换的，因为它们可能还会被频繁访问。/忽视最近趋势：MFU 不会很好地适应最近的使用模式变化

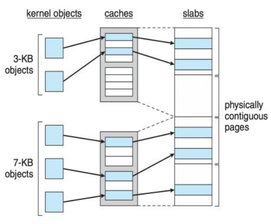


Page-Buffering Algorithms:Keep a pool of free frames, always(• frame available when needed, no need to find at fault time • Read page into free frames without waiting for victims to write out • Restart as soon as possible • When convenient, evict victim); Possibly, keep list of modified pages(When disk idles, write pages there and set to non-dirty: this page can be replaced without writing pages to backing store)
Frame Allocation: Equal allocation(• For example, if there are 100 frames and 5 processes, give each process 20 frames • Keep some as free frame buffer pool) Proportional allocation(• Allocateaccording to the size of process • Dynamic as degree of multiprogramming, process sizes change) Global vs. Local Allocation. 全局替换优点：灵活,进程不受限于只能使用自己的帧；吞吐量提升,内存资源被更充分地利用。缺点：执行时间变化,进程可以被迫放弃其帧给其他进程，因其他进程的行为而发生较大波动；资源竞争,引起更频繁的页面替换 局部替换优点：一致性：每个进程的性能更加稳定和可预测，不受其他进程内存使用影响；简化管理,每个进程只管理自己的帧集。缺点：物理内存的使用效率不高，尤其当某些进程不频繁访问其分配的内存时；吞吐量降低
全局替换更适合于资源共享密集且需要高吞吐量的环境，但可能导致单个进程性能的不确定性;局部替换在确保每个进程性能稳定性方面表现更好，但可能不充分利用可用的物理内存资源。
Reclaiming Pages: A strategy to implement global page-replacement policy,Rather than waiting for the list to drop to zero before we begin selecting pages for replacement, page replacement is triggered when the list falls below a certain threshold.→ attempts to ensure there is always sufficient free memory to satisfy new requests
Major and minor page faults. Major: page is referenced but not in memory(• Can only be satisfied by disk • do_anonymous_page is not major) Minor: mapping does not exist, but the page is in memory (• Shared library • Reclaimed and not freed yet)
Non-Uniform Memory Access: 每个 CPU 都可以比其他 CPU 更快地访问内存的某些部分; Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled (modifying the scheduler to schedule the thread on the same system board when possible) Linux 中的实现: Linux 内核维护调度域，这些域定义了线程可以迁移的范围。内核不允许线程跨域迁移; Linux 可以为每个 NUMA 节点维护一个独立的空闲帧列表，从而确保从运行线程的当前节点分配内存; Linux 支持 NUMA 感知的内存分配，可以根据线程当前所在的节点来优化内存分配。
Thrashing:一个进程可用的帧数量少于其频繁访问的页面数目会频繁出现 page fault；同一个 page 被频繁地换入换出，其调页时间甚至会大于执行时间 Option I:using local page replacement(One process starts thrashing does not affect others) Option II:Provide a process with as many frames as it needs
Working-Set Model: Working-set window(Δ): a fixed number of page references(• if Δ too small ➠ will not include entire localitym• if Δ too large ➠ will include several localities • if Δ =∞ ➠ will include entire program) Working set of process pi (WSSi): total number of pages referenced in the most recent Δ (varies in time) Total working sets: D = ∑ WSSi(• approximation of total locality • if D > m ➠ possibility of thrashing • to avoid thrashing: if D > m, suspend or swap out some processes)
Keeping Track of the Working Set: 使用间隔定时器(每隔一定的时间单位发出中断)和一个引用位(在访问页面时置 1) EX:工作集窗口设定为 Δ = 10,000，每隔 5000 时间通中断更新页面的状态,为每个页面在内存中保留两个位（比如称为位 0 和位 1）来记录其引用状态。• 每当中断发生时遍历所有页面 • 对于每个页面，操作系统将其引用位复制到一个内存位中（位 0 或位 1），然后将引用位重置为 0 • 交替使用两个内存位来记录连续的两个时间间隔的页面访问状态。• 判断页面是否在工作集中：如果任何一个内存位为 1，表明在最近的 10,000 时间，页面至少在一个时间间隔内被访问过，因此它被认为是工作集的一部分。
Page-Fault Frequency(PFF)策略比 WSS 更直接，设定一个页面错误率的阈值，一个进程的实际 rate 低于阈值，分配了过多的内存,减少分配给该进程的内存帧数目；一个进程 rate 高于阈值,需要更多的内存,增加分配给该进程的内存帧数目；如果没有空闲帧可用，需要 swap out 某个进程；优点:根据每个进程的实际内存使用情况动态调整内存分配，这有助于优化内存利用率和系统性能。缺点：频繁调整内存分配导致较高的管理开销
Kernel Memory Allocation:allocated from a free-memory pool(• Kernel 数据结构大小区分比较大，很多小于 page-> 节省内存，努力减少碎片 • Some kernel memory needs to be physically contiguous); Other Considerations:• Prepaging • Page

size • TLB reach • Inverted page table • Program structure • I/O interlock and page locking

Prepaging:To reduce the large number of page faults that occurs at process startup(• Prepage all or some of the pages a process will need, before they are referenced • But if prepaged pages are unused, I/O and memory was wasted) Page Size. Page size selection must take into consideration:(• Fragmentation -> small page size • Page table size -> large page size • Resolution -> small page size(小页面提供了更细粒度的内存管理) • I/O overhead -> large page size • Number of page faults -> large page size • Locality -> small page size(常用的数据不太可能与频繁访问的数据在同一页面上) • TLB size and effectiveness -> large page size) TLB Reach = (TLB size) × (page size),Increase the page size to reduce TLB pressure I/O interlock:Pages must sometimes be locked into memory(当从一个设备复制文件到另一个设备时，用于这一操作的内存页面需要在整个复制过程中保持锁定)

Linux Buddy System: 从物理连续的段上分配内存；每次分配内存大小是 2 的幂次方，例如请求是 11KB，则分配 16KB; 当分配时，从物理段上切分出对应的大小



Slab Allocator: a cache of objects, 过程: 当内核需要一个新的数据结构时,从相应数据结构的缓存中的一个 Slab 分配一个对象,如果一个 Slab 的所有对象都被使用了（即满了），则从另一个空的或新创建的 Slab 分配对象,当对象不再需要时，它会被返回到 Slab，并标记为可用. 优势:减少碎片化：通过重用相同大小和类型的对象减少内存碎片/快速分配：由于对象都是预分配和预初始化的，内存分配速度更快/部分初始化：一些对象字段可能是可重用的，这意味着在重新分配对象时不需要完全初始化。 Slab Allocator in Linux: 当需要创建一个新的任务时，系统从专门用于 task_struct 的缓存中分配一个新的结构体。Slab 的状态:• Full – all used • Empty – all free • Partial – mix of free and used

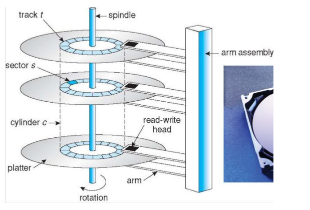
分配过程:使用 Partial Slab,首先尝试从处于部分使用状态的 Slab 中分配一个空闲对象→使用 Empty Slab：如果没有部分使用的 Slab，那么尝试从完全空闲的 Slab 中分配→创建新 Slab：如果没有空闲的 Slab 可用，系统将创建一个新的空 Slab，并从中分配对象。

SLAB 分配器(Linux 2.2 引入):为每种内核数据结构维护一个缓存，每个缓存包含了多个 Slab，每个 Slab 包含了多个相同大小和类型的对象。

SLOB 分配器(适用于内存有限的系统):使用一个较为简单的方法来管理内存，它维护了三个列表，分别用于小、中、大尺寸的对象。。

SLUB 分配器:去除了 SLAB 中的 per-CPU 队列，并将元数据直接存储在页面结构中，以减少管理开销。

How kernel manage physical RAM: 识别可用内存、跟踪使用情况、分配和释放内存等; 分配页给进程。将进程的虚拟内存地址映射到物理内存地址



Disk Structure：Platter 表面被分成了很多环形 track（磁道），再细分为 sector（扇区）。在一个 arm position 下的 track 组成一个 cylinder（柱面）

Positioning time：time to move disk arm to desired sector=seek time(move disk to the target cylinder) + rotational latency(for the target sector to rotate under the disk head); average latency = 1/2 latency

Average access time = average seek time + average latency ; Average I/O time = average access time + (data to transfer / transfer rate) + controller overhead e.g., to transfer a 4KB block on a 7200 RPM(每分钟转 7200 圈) disk; 5ms average seek time, 1Gb/sec transfer rate with a 0.1ms controller overhead = 5ms + 4.17ms + 4KB / 1Gb/sec + 0.1ms (4.17 is average latency)

Disk Scheduling: chooses which pending disk request to service next (to minimize seek time,rotational latency is difficult for OS to calculate)

FCFS: Advantage: • Every request gets a fair chance • No indefinite postponement; Disadvantages:• Does not try to optimize seek time• May not provide the best possible service

SSTF:shortest seek time first. Advantage: •Average Response Time decreases •Throughput increases; Disadvantages: •Overhead to calculate seek time in advance •Can cause Starvation for a request if it hashigher seek time as compared to incomingrequests • High variance of response time as SSTFfavors only some requests

SCAN:从一边往另一边走，选择路上的 request 不回头，直到到达另一端反向. Advantages: •Average Response Time •High Throughput •Low variance of response time Disadvantages:•Long waiting time for requests for locations just visited by disk arm

C-SCAN:scan + 到达一端时立刻回到开头 Advantage: Provides more uniform wait time compared to SCAN

LOOK/C-LOOK:在 SCAN / C-SCAN 的基础上，只走到一端最后一个任务（look 是否有请求）而不走到头.advantage:Prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Selecting Disk-Scheduling Algorithm:SSTF 最常见，当 I/O 负荷较大时,使用 LOOK / C-LOOK 避免 starvation。(文件系统如果注重空间局部性，能够提供很好的表现提升)

Nonvolatile Memory Devices: SSD(Solid-State Drives)：使用 Flash Memory 存储数据;USB ：如 U 盘（Thumb Drive）或闪存驱动器（Flash Drive），便携式且容易使用;DRAM 磁盘替代品：DRAM 技术提供极高速度，需要电源保持数据。优势:比硬盘驱动器（HDD）更可靠;更快的数据访问速度，无寻道时间或旋转延迟;可以直接连接到 PCI 总线等高速接口。;不存在机械延迟，简单的调度策略（如首到先服务）通常非常有效。成本和寿命,成本：每 MB 的成本高于传统硬盘; 寿命：某些类型的非易失性存储设备（如 NAND 闪存）可能有写入次数限制;容量通常低于传统硬盘。

NAND Flash Controller Algorithms:闪存转换层（FTL）表:用于跟踪哪些逻辑块是有效的,FTL 表将逻辑块地址映射到物理块地址; Garbage Collection 清理这些无效数据，首先识别包含无效数据的块，将其中有效的数据复制到其他地方,然后清除整个块，使其恢复到可写状态; Over-Provisioning 提供额外的存储空间，用于垃圾回收和负载均衡;

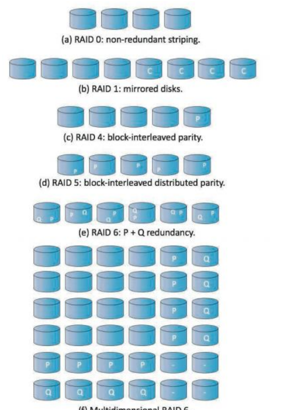
Magnetic Tape: 特点:大容量; 慢速访问时间(随机访问慢);寻道时间更长;数据位于读写头下，传输速率快,需要卷带定位，整体数据访问效率较低。;不适合频繁的随机访问;存储的数据相对稳定，不易丢失，适合长期存储。

Disk Management.Physical Formatting：将磁盘划分成扇区，以便控制器可以读写;Partitioning:将磁盘划分为多个分区，每个分区包含一组柱面;Logical Formatting:在磁盘分区上创建文件系统的过程; clusters：能将多个块组合成一个更大的单元; Root partition:根分区通常包含操作系统(• Mounted at boot time • Other partitions can mount automatically or manually),其他分区用于存储其他操作系统、不同的文件系统; Boot Block：引导块包含启动计算机所需的初始代码; Raw Disk Access: 某些应用程序，特别是数据库需要直接管理磁盘的数据块，绕过操作系统的文件系统，直接与磁盘上的数据块进行交互; Bootstrap: 引导程序通常存储在固件或 ROM 中，负责硬件的初始化和加载更高级别的引导加载程序; Bootstrap Loader 存储在启动分区的引导块中,负责加载操作系统核心到内存中，启动操作系统。

Disk Attachment: host-attached storage; network-attached storage; storage area network

RAID: Disks are unreliable, slow, but cheap, so use redundancy(• Increases reliability • Increases speed)

RAID 技术. • Data Mirroring:Keep the same data on multiple disks; • Data Striping: Keep data split across multiple disks to allow parallel reads; • Error-Code Correcting (ECC) - Parity Bits: Keep information from which to reconstruct lost bits due to a drive failing



(a) RAID 0: non-redundant striping.
(b) RAID 1: mirrored disks.
(c) RAID 4: block-interleaved parity.
(d) RAID 5: block-interleaved distributed parity.
(e) RAID 6: P + Q redundancy.
(f) Multidimensional RAID 6.

RAID 0:数据被均匀 striped 多个磁盘上; 无冗余：不提供奇偶校验或镜像功能，如果任何一个磁盘失败，所有数据都可能丢失; 固定条带大小; 读写性能提升：允许同时从多个磁盘读写数据; 单条带访问速度：访问单个带速度并不比访问单个磁盘快，性能提升来自于能够并行处理多个条带。不提高可靠性：不提供数据冗余，因此它不增加数据的可靠性; 适用场景:对高性能有要求，但数据可靠性要求不高的场景。例如，它可以用于高速缓存、临时文件存储等场合

RAID 1: 数据镜像：数据被完全复制到两个磁盘上。每次写入操作会同时在镜像磁盘上执行; 冗余性：很高的数据冗余和可靠性。一块磁盘故障，数据仍然可以从另一块磁盘上恢复; 磁盘使用效率：因此存储效率为 50%。性能: 写入性能受限于最慢的磁盘，每个写操作需要在两个磁盘上同时完成/读取性能可以提升,从两个磁盘中选择寻道时间短的读取数据; 适用场景: 对数据安全性有高要求的应用场景。如重要的数据库、关键系统文件等。

RAID 2: 在位级别对数据进行分割，使用 Hamming 码进行错误纠正。

RAID 3: 位级别奇偶校验, 使用一块专用的奇偶校验磁盘，用于存储所有数据磁盘的奇偶校验信息。使用奇偶校验比特来进行数据恢复。EX: 一个具有 4 个数据磁盘和 1 个奇偶校验磁盘的 RAID 3 阵列存储了比特序列 "0110"，奇偶校验比特将存储所有数据比特的异或值(((0 xor 1) xor 1) xor 0) = 0。如果丢失了比特 "0"，可以运算：(((0 xor 0) xor 1) xor 0) = 1。恢复数据。性能(缺点是数据恢复需要较长的时间，因为涉及到多次比特级别的异或运算)

RAID 4：类似于 RAID 3，但在数据块之间进行交错,每个读取操作只涉及一个磁盘，因此具有较低的读取延迟,使用单个奇偶校验磁盘来存储奇偶校验信息。

RAID 5：类似于 RAID 4，但奇偶校验信息分布在所有磁盘上，而不是只有一个奇偶校验磁盘,具有更好的容错性，因为奇偶校验信息分布在多个磁盘上，可以容忍多个磁盘的故障。,读取性能较高，因为小读取操作可以从多个磁盘并行读取。

RAID 6：在 RAID 5 的基础上增加了一个额外的奇偶校验块，从而提供更高级别的容错性。具有双奇偶校验块，可以容忍两个磁盘的故障，而 RAID 5 只能容忍一个。更高的容错性，但写入性能较低，因为需要计算和写入两个奇偶校验块。

Polling: For each I/O operation: • busy-wait if device is busy (status register)(Cannot accept any command if busy) • send the command to the device controller (command register) • read status register until it indicates command has been executed • read execution status, and possibly reset device status • Polling requires busy wait(reasonable if device is fast; inefficient if device slow)

Interrupts: Interrupts can avoid busy-wait • device driver (part of OS) send a command to the controller (on device), and return • OS can schedule other activities • device will interrupt the processor when command has been executed • OS retrieves the result by handling the interrupt(Interrupt-based I/O requires context switch at start and end so if interrupt frequency is extremely high, context switch wastes CPU time)

Direct Memory Access: DMA transfer data directly between I/O device and memory(• OS only need to issue commands,a command includes: operation, memory address for data, count of bytes···, data transfers bypass the CPU • no programmed I/O (one byte at a time), data transferred in large blocks • it requires DMA controller in the device or system)

Application I/O Interface: I/O 系统相关的系统调用将不同的 I/O 设备的工作方式封装到一些类中，从而形成较少的通用类型，从而为应用程序隐藏硬件的具体差异(in Linux, devices can be accessed as files; low-level access with ioctl) Data transfer mode •character：逐个字节传输（如 terminal）•block：以块为单位传输（如 disk）Access method•sequential（如 modem）•random（如 CD-ROM）Transfer method•synchronous：需要按预计的响应时间进行传输并和系统的其他方面相协调（如 keyboard）•asynchronous：响应时间不需要规则或者可预测，不需要与其他计算机事件协调（如网络 I/O）Sharing•sharable：可以被多个进程或线程并发使用（如 keyboard）•dedicated：不能（如 tape）

Kernel I/O Subsystem: I/O scheduling(• to queue I/O requests via per-device queue • to schedule I/O for fairness and quality of service); Buffering - store data in memory while transferring between devices(• to cope with device speed mismatch: receive from network and write to ssd, double buffering • to cope with device transfer size mismatch: network buffer reassembly of message)

Caching: hold a copy of data for fast access Spooling: A spool is a buffer that holds the output (device's input) if device can serve only one request at a time i.e., printing Device reservation: provides exclusive access to a device(• system calls for allocation and de-allocation • watch out for deadlock)

Improve Performance(• Reduce number of context switches • Reduce data copying • Reduce interrupts by using large transfers, smart controllers, polling • Use DMA • Use smarter hardware devices • Balance CPU, memory, bus, and I/O performance for highest throughput • Move user-mode processes / daemons to kernel threads)

File System: abstraction of disk(File→Track/sector)，文件系统由两个部分组成：文件 (file) 的集合，每个文件存储一些数据，是存储的逻辑单位; 目录结构 (directory structure)，用于组织统内的文件并提供相关信息

File Concept: File is a contiguous logical space for storing information(在 Unix 中就是一块连续的字节); 文件是用户和程序用来存储和获取信息的途径。从用户角度来看，文件是逻辑外存的最小配单元,数据只有通过文件才能写到外存; 文件有不同的类型，例如数据文件(character, binary, and application-specific)、可执行文件等。

File Attributes. •Name:这是唯一的以 human-readable 形式保存的信息。•Identifier:一个在当前文件系统中唯一的 tag（通常为 number），文件系统用它来标识文件。•Type:一些文件系统支持不同的文件类型。•Location:标识文件在哪个设备的哪个位置。•Size:当前文件大小，也可能包含文件允许的最大大小。•Protection:Access-control information。•Timestamp:保存创建时间、上次修改时间、上次使用时间等。•User indentification:创建者、上次修改者、上次访问者等。

File Operations. • create: space in the file system should be found; an entry must be allocated in the directory • open: most operations need to file to be opened first, return a handler for other operations • read/write: need to maintain a pointer • reposition within file – seek:将 current-file-position pointer 的位置重新定位到给定值，例如文件开头或结尾。 • close • delete:Release file; space; Hardlink: maintain a counter - delete the file until the last link is deleted • truncate: empty a file but maintains its attributes

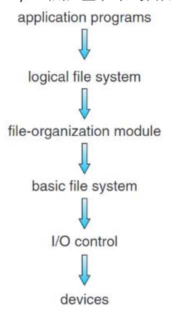
Open Files. Several data are needed to manage open files: • Open-file table: tracks open files; • File pointer: pointer to last read/write location, per process that has the file open • File-open count: counter of number of times a file is open→to allow removal of data from open-file table when last processes closes it • Disk location of the file: cache of data access information • Access rights: per-process access mode information

file lock 共享锁 (shared lock): multiple processes can acquire the lock concurrently; 独占锁 (exclusive lock): one process can acquire such an lock 2 种机制，强制锁定 (mandatory lock)，即一旦进程获取了独占锁，操作系统就阻止任何其他进程访问对应文件，即操作系统本身确保了完整性；建议锁定 (advisory lock)，即进程可以自己得知锁的状态然后决定要不要坚持访问。Windows 使用前者，UNIX 使用后者。

File Types. 方式 1：文件扩展名 (file extension)；例如规定只有扩展名是 .com, .exe, .sh 的文件才能执行; 方式 2：通过在文件开始部分放一些 magic number 来表明文件类型; 方式 3: 操作系统不试图识别文件的类型,当用户尝试运行文件,会照常按照预期的编码方式解析

File Structure. 1.无结构 (no structure) 文件: a stream of bytes or words 2. simple record structure 文件，以 record 为单位，每个 record 可以是定长或者变长的(数据库) 3. complex structures 文件。例如 Microsoft Word 文档

Access Methods. •Sequential access(a group of elements is access in a predetermined order) • Direct access(access an element at an arbitrary position in a sequence in (roughly) equal time, independent of sequence size) •Indexed Sequential-Access(即像字典那样确定所需访问的内容在哪一块，再在对应块中顺序寻找)



Directory Structure:实现新建、删除、查找、遍历、列出目录中的文件等基本操作;兼顾效率、便于使用、便于按一些属性聚合等要求 Single-Level Director: 所有文件都包含在同一目录中; 问题:当文件数量增加时，没有有效的方法来查找或者给文件分组;名称必须是唯一的。在有多个用户时想要同一个名字 Two-Level Directory：为每个用户创建自己的 用户文件目录 (User File Directory, UFD)；这些 UFD 聚合成为主文件目录(Master File Directory, MFD)，当用户登录时，操作系统从 MFD 中找到用户的 UFD Tree-Structured Directories: 目录本身也被放在父目录的一个条目中；每个条目有一个 bit 来表示这个条目表示的是一个文件还是一个子目录；每个进程有一个当前目录 (current directory)；当用户指定一个文件名时，就在当前目录中寻找。用户也可以通过系统用 change_directory() 来更改当前目录；路径名 (path name):绝对路径名(表示从根开始到指定文件的路径)/相对路径名(表示从当前目录开始的路径); 当删除一个目录时，一种设计思路是不允许删除非空的目录；另一种是删除目录下的所有文件（包括子目录）；无法共享 Acyclic Graph Directories：在 Tree-Structured Directories 的基础上支持了目录共享子目录或者文件; 实现方式：用链接文件来记录指向文件的绝对或相对路径,当访问这种文件时，操作系统通过该路径名来 resolve 链接，从而定位真实文件; 问题: 如果删除一个被某个 soft link 所引用的文件，会留下 dangling pointer→方法 1：当真实文件被删除时，不处理这些 link,用户在试图访问这些 link 时会发现真实文件已经被删除/方法 2: 复制被引用文件的所有信息,引入引用计数 (reference counter),在每个文件被创建一个新的硬链接时 +1,而删除一个硬链接或原文件本身时 -1,当计数为 0 时，文件可以删除。 General Graph Directory:允许目录中有环；使用 垃圾回收 (garbage collection) 方案来确定哪些文件可以删除

Hard link & soft link 1.Hard link 是一个 directory entry; soft link 是一个 file 2. Hard link 不能引用 directory;soft link 可以(每个目录中的 file name . 是指向自己的 hard link, ..是指向父目录的 hard link)
3. Hard lin 不能跨越 file-system boundaries, soft link 可以

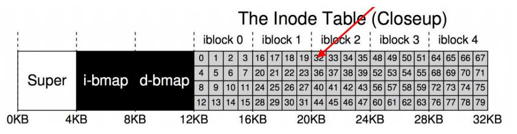
File System Mounting:A file system must be mounted before it can be accessed(• mounting links a file system to the system, usually forms a single name space • the location of the file system being mounted is call the mount point • a mounted file system makes the old directory at the mount point invisible,当文件系统挂载到目录（挂载点）时，它会遮盖该目录中现有的任何文件和子目录。这些原始内容并没有被删除，而是暂时无法访问；它们被隐藏在挂载的文件系统后面。一旦文件系统卸载，目录的原始内容将再次变得可见)

Protection:给每个文件和目录维护一个 Access Control List (ACL)，指定每个用户及其允许的访问类型。Advantages: fine-grained control; Disadvantages: How to construct the list/How to store the list in directory. UNIX 将用户分为了三类（owner, group, others)，并将文件访问也分为了三类 (read, write, execute)：

File-System Structure: Application programs 给出 read / write / open 指令 logical file system: 管理所有文件系统所需的 metadata（即除了文件内容之外），例如 directory structure。通过 File Control Block (FCB) 保存 name, ownership, permissions 等; 将上层给出的 read / write / open 指令解析为 read / write 某些 logical blocks 的指令

File-organization module 负责 logical file blocks 到 physical file blocks 的转换;负责管理 free space,跟踪未被使用的 blocks 在需要时进行分配; 将 logical blocks 指令转换为 physical file blocks

Basic file system 管理包含 file-system, directory 和 data blocks 的缓存提高性能;如果上层指令 miss,将指令传递给 I/O control I/O control 包括设备驱动程序和中断处理程序，以在主存和磁盘系统之间传递信息; 将上层的指令转换为 low-level, hardware-specific 的指令来实现相关操作
分层的文件系统设计可以降低文件系统本身的复杂性和冗余性，但是会影响性能
File System Data Structures. On-disk structures: • File control block(FCB) (per file):保存 name, ownership, permissions, ref count, timestamps, pointers to data blocks on disk 等信息；每个 FCB 有一个唯一的标识号,在 NTFS 中,每个 FCB 是一个叫 master file table 的结构的一行； • Boot control block (per volume):如果 volume 是 boot partition 保存从该 volume 引导操作系统所需的信息;否则内容为空,通常是 volume 的第一个 block,UFS 称之为 boot block,NTFS 称之为 partition boot sector; • Volume control block (per volume):包含当前 volume 的 block #, block size, free-block #, free-block ptrs, free-FCB #, free FCB ptrs 等,UFS 称之为 superblock,NTFS 将其存在 master file table 中; • Directory (per FS):组织文件,包含文件名和对应 FCB 的标识号,在 NTFS 中，被存在 master file table 中
In-memory structures: • Mount table:每个挂载的 volume 占一行,保存相关信息; • Directory cache; • Global (a.k.a. system-wide) open-file table; • Per-process open-file table; • 读写磁盘的 buffer
File Creation:当创建新文件时，application programs 调用 logical file system, 后者分配一个 FCB。系统随后将对应的 directory 读到内存,并用 filename 和 FCB 更新 directory。
File Open & Close:系统调用 open() 将文件名传给 logical file system,后者搜索 system-wide open-file table 以确定该文件是否正在被其他进程使用：如果不是，在 directory 中找到这个 file name，将其 FCB 从磁盘加载到内存中，并将其放在 system-wide open-file table 中。然后,在当前进程的 per-process open-file table 中新建一个 entry 指向 system-wide open-file table 中的对应项。如果是,则直接在当前进程的 per-process open-file table 中新建一个 entry,指向 system-wide open-file table 中的对应项即可。这可以节省大量开销。最后,open()返回指向 per-process open-file table 中对应 entry 的指针,文件操作通过这个指针执行。这个 entry 在 UNIX 中被称为 file descriptor，在 Windows 中被称为 file handle。
当进程关闭文件时，per-process open-file table 中对应 entry 将被删除,system-wide open-file table 中的 counter 将被 -1；如果该 counter 清零,则更新的 metadata 将被写回磁盘上的 directory structure 中，system-wide open-file table 的对应 entry 将被删除。
Mounting File Systems：引导块（Boot Block）：设备上的一系列顺序块组成,包含了一个名为 boot loader 的内存映像,定位并挂载存储设备的 root partition。root partition 包含操作系统的内核。
内存中的挂载表（In-memory Mount Table）：保存每个文件系统附加到系统目录结构的挂载点; 挂载的文件系统类型; 指向每个挂载文件系统的访问路径.
Unix 和内存中的挂载表包含了指向该文件系统 superblock 的指针(superblock 它保存了有关文件系统的元数据，如其大小、类型、状态和其他控制结构（如索引节点）的信息)
Virtual File Systems:同时支持多种类型的文件系统,操作系统定义通用的文件系统访问接口，即 open(), read(), write(), close() 和 file descriptors；所有文件系统都需要实现它们，并把它们按照约定排布在函数表中。VFS 将上述通用操作和实现分开，即在需要调用某个函数时，去对应 FS 的函数表的约定位置找到函数指针就可以访问了。
Linux defines four VFS object types:• superblock: defines the file system type, size, status, and other metadata• inode: contains metadata about a file (location, access mode, owners…)• dentry: associates names to inodes, and the directory layout• file: actual data of the file
Directory Implementation. 目录是一种特殊的文件,保存 file name 到 FCB 的映射关系; 实现: linear list(费时);有序表、平衡树、B+树、hash table
Disk Block Allocation. Contiguous Allocation:每个文件在磁盘上占有的 blocks 是连续的; 优点：降低 seek time、目录只需要维护文件的起始 block 及其长度; 缺点: 外部碎片、文件是会扩展的(确定一个文件需要多少空间); 方法 1: 将文件复制到一个新的足够大的空闲空间里去(overhead 大) 方法 2:用户确定文件的最大大小(不便，而且用户可能高估所需空间导致内部碎片) 方法 3: 创建连续空间的链表来保存文件。当空间不够时，添加一块连续空间（extent）。FCB 记录的信息除了起始和长度外,还额外维护下一个 extent 的信息 Linked Allocation:文件是 block 组成的链表;每个 block 中保存下一块的地址;目录只需要记录起始地址和结束地址; 优点: 没有外部碎片; 问题:获取一个文件需要更多的 I/O 和 disk seek; 每个块中有 overhead; 方法: 将多个块组成 cluster。(增加内部碎片) 问题: pointer 损坏. 方法: 通过冗余信息（如双向链表），但带来额外开销。问题:不支持 random access Indexed Allocation:给每个文件记录一个 索引块 (index block)，记录这个文件的第 i 个块在磁盘中的哪个块。目录只需要保存索引块放在哪里; 优点: 支持 random access,不会有外部碎片; 缺点: 索引块开销、允许文件中存在 holes;
实现: linked index blocks 每个 index block 占用一个 disk block 形成一个 disk block 的链表; multiple-level index blocks 例如，每个 disk block 为 4KB，一级索引有 1024 个 4B 的指针指向二级索引，每个二级索引有 1024 个 4B 的指针指向文件块，因此就能够有 2^20 块 comblined scheme 在 inode 中有 15 个指针（假设 block size = 512B, ptr = 4B）：前 12 个指向 direct block，如果文件小于 12 * 512B = 6KB，不需要 index block；第 13 个指向 single indirect block, 索引 128 * 512B = 64KB 数据; 第 14 个指向 double indirect block,索引 128 * 128 * 512B = 8MB 数据; 第 15 个指向 triple indirect block,可以索引 128 * 128 * 128 * 512B = 1GiB 数据
Free-Space Management. 文件系统需要维护 free-space list 以得知空闲的块或 clusters 方法 1：bitmap 每个 block 用 1 bit 空闲或占用,为了减少 bitmap 占用的空间，可以以 cluster 为单位 方法 2: 所有的 free space 用链表连接(在需要多个空间块时需要比较多的 I/O，效率较低) 方法 3: grouping 维护若干个 block 形成的链表，每个 block 保存若干空闲块的地址减少需要的 I/O
counting 维护连续的空闲块的链表,链表的每个结点是连续的空闲块的首块指针和连续的长度
File System Performance •Disk allocation and directory algorithms•Types of data kept in file's directory entry•Pre-allocation or as-needed allocation of metadata structures•Fixed-size or varying-size data structures→To improve file system performance:• Keeping data and metadata close together•Use cache: separate section of main memory for frequently used blocks• Use asynchronous writes, it can be buffered/cached, thus faster(Cannot cache synchronous write, writes must hit disk before return; Synchronous writes sometimes requested by apps or needed by OS) •Free-behind and read-ahead: remove the previous page from the buffer, read multiple pages ahead
File System Interface:O_CREAT: create the file if not exists; O_WRONLY: can only write to the file;O_TRUNC: if the file exists, truncate it to zero bytes; write():Write this data to persistent storage, at some point in the future; fsync(): forces all dirty (not yet written) data to disk
Why removing a file calls unlink:Create a filelinking a human-readable name to that file (or inode), and putting that link into a directory
Inode address calculation:每个 inode 为 256 字节;每个块大小为 4 KB 每个块中的 Inode 数量=16 个 inodes,如果用 5 个块来存储 inodes，那么总 inodes 数量= 5* 16=80 个 inodes,代表 80 个文件或目录。计算 inode 32 的地址：字节偏移量=32×256= 8 KB 加上 inode 块之前其他组件的偏移量：+super block 4 KB,BITMAP 8 KB
→4 KB+8KB+8 KB=20 KB



Security in OS: CPU privileged mode; Memory partition and paging for memory isolation; IO access control list
Trusted Computer System Evaluation Criteria (TCSEC):Four division: D – Minimal protection; C – Discretionary protection; B – Mandatory protection; A – Verified protection
Concepts in system security: Trusted Computing Base(为实现计算机系统安全保护的所有安全保护机制的集合包括软件、硬件和固件); Attacking Surface(一个组件被其他组件攻击的所有方法的集合,可能来自上层、同层和底层); Defense in-depth(为系统设置多道防线，为防御增加冗余，以进一步提高攻击难度)Trusted computing base: The larger the TCB is, the more difficult it is to make it secure.;The TCB should be as simple as possible(Minimize misuse/ Enable better verification); Since a component almost always depends upon its lower levels for security, the TCB usually includes all lower levels.(The lower the level that protections can be added the smaller the TCB.The smaller the TCB, the easier it is to validate)Attacking Surface:The smaller the attack surface, the easier it is to protect.Defense in-depth: Its intent is to provide redundancy in the event a security control fails or a vulnerability is exploited
Access control:Access control is an essential element of security that determines who is allowed to access certain data, apps, and resources—and in what circumstances. Authentication :验证某个发起访问请求的主体的身份; Authorization :授予某个身份一定的权限以访问特定的对象; Auditing: record what users and programs are doing for later analysis
ACLs:Can have large numbers of objects;Easy to grant access to many objects at once;Require expensive operation on every access Capabilities:Hard to manage huge number of capabilities;They have to come from somewhere;They are fast to use (just pointer dereferences) →Most systems use both: ACLs for opening an object (e.g. fopen()); Capabilities for performing operations (e.g. read())