Operating system

Chapter notations may vary from prof. presentations

2. The kernel abstraction

Booting

- BIOS copies bootloader //
- Passes execution to bootloader.

Anytime we execute (any) instructions, generally happens from physical memory, because it is much faster than fetching from disc.

- Bootloader copies OS kernel, and passes instructions/executions to CPU
- The OS then finishes with logging instructions

Protection

Execute code with restricted privileges, some challenges may be that the code is buggy or malicious. We don't want a process restricting other processes, monopolize hardware, affect OS or see protected data structures in memory.

Want to restrict this process so it only has access to a limited number of interfaces.

One way we can do this is by executing in a simulator first and checking if the program tries to do something it does not have privileges to. Somewhat slower because the another system has to check all executions in order to open it. A faster method is to run the unprivileged code directly on the CPU. Most instructions are safe, some has to be checked such as reading/writing and opening files.

2.1 Dual mode operation

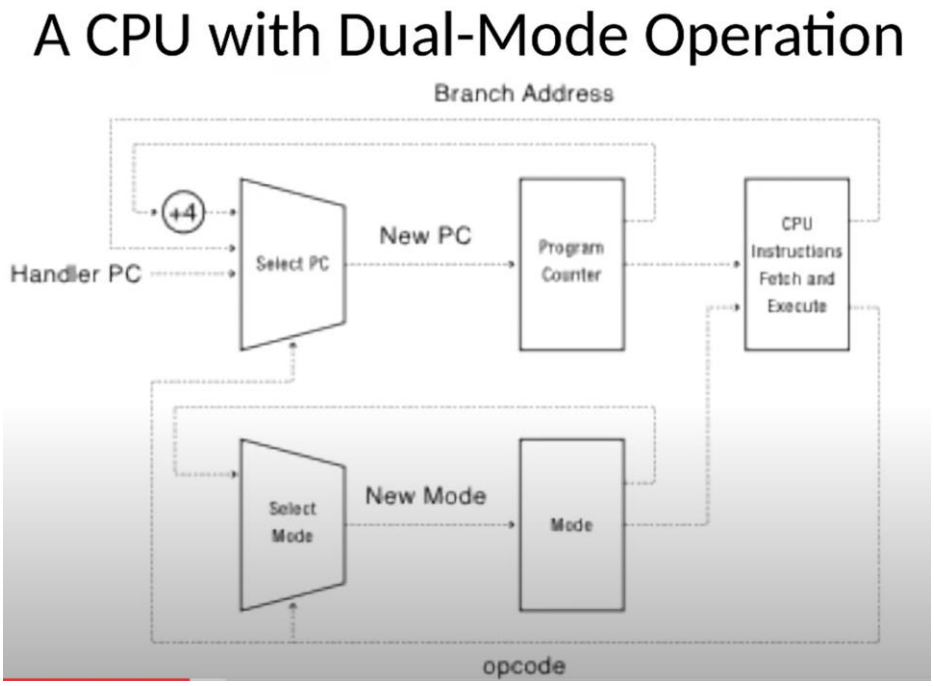
Dual mode operation essentially breaks down executions on the CPU into two different modes.

- The kernel mode can execute with complete privileges "god mode". Can read/write to any memory, disk sector, access I/O devices, read any packets.
- User-mode: execute with limited privileges. These privileges are only granted by the OS kernel.

On the x86, the user/kernel mode are stored in the EFLAGS register that maintains when and what an execution has privileges to do. The eflags register tells the cpu which mode we are in. If we are in user mode and we try reading from kernel memory, then we would have a processor exception and the control would be passed to an execption handler. (Researching this, I found that eflag register does not contain these said values, rather the eflags register has condition codes that are modified as a by-product of executing instructions; the eflags register also has other flags that control the processor's behavior such as whether interrupts are masked or not.)

We can change between these two modes through a safe control transfer. For example, the only way to access hardware is through the kernel. Control has to be transferred on the CPU from the user process to the kernel instructions.

How do we do this?



In terms of the hardware of a dual mode operation, we need privileged instructions that are available to the kernel only, limits on memory access to prevent user mode to overwrite the kernel, a timer that works as a “second life” to regain control from a user program in a loop and back to the kernel (it cannot be overwritten), and lastly a method to incorporate the dual mode switching.

2.1.1 Privileged instructions

The user application has to ask the kernel to operate on behalf of the user program (when in user mode). The user program should not be able to change the set of memory addresses it can access.

Processor exception is a memory location predefined that can pick up execution and kill the program that attempted to execute an illegal executions and pass the execution back. Gives the cpu back to the kernel.

Source code is compiled and turned into instructions and data that the os copies. This is then saved in the physical memory where machine instructions processes the data. Heap contains dynamically allocated data, whereas the stack contains often used data. These two datastructures grow and shrink. The os kernel is totally separated or isolated from this process.

Process abstraction

Process : an instance of a program, running with limited rights. A process contains an address space which is a set of rights of a process. In addition to this we also have to set up a process control block. This is a part of a data structure in the OS, that keeps track of all the permissions and open files that are associated with a process.

- A thread is a sequence of instructions within a process.

2.1.2 Memory protection

Need good memory protection in order to limit the memory regions ta process can access.

A simple approach to providing protection would be to introduce two extra registers in the processor; a base and a bound. The base specifies the start of the process's memory region and the bound, the endpoint.

When a processor requests either read or write from memory, we are given a physical address that corresponds to somewhere in our physical memory. That physical address contains a scope that our process can access, that is defined by a base and a bound address. If we are not in this memory region, say a process tries to access memory region 512, whereas it has been given the base is from 1024-2048, it would raise an exception, and control would be transferred back to the kernel.

However using this simple approach would result in us missing out in some important features.

1. **Expandable heap/stack:** With a single pair of base and bound registers to each process, the amount of memory would be fixed when the process starts. This would prevent memory regions to independently expand due to program behaviour. The heap contains dynamically allocated objects and the stack, procedure local variables.
2. **Memory sharing:** Base and bound registers do not allow memory to be shared between different processes (copy data between programs through os, difficult to collaborate).
3. **Physical memory addresses:** When a program is compiled and linked, the addresses of its procedures and global variables are set relative to the beginning of the executable file. Since a program may be loaded at different locations, the kernel must change instructions and data locations.
4. **Memory fragmentation:** Once a program starts, it is nearly impossible to relocate it. Memory will become increasingly fragmented due to irregularities when it comes to application start and finish times. Base-limit also restricts the physical memory for other processes running in the same time preventing a contiguous memory, so called memory fragmentation.

A solution to these given challenges is introducing virtual memory. Address spaces are mapped to a table that are again mapped to physical memory. This ccreates the illusion that a process has the

entire machine to itself. The table is set up by os kernel. The layer of indirection provided by virtual addresses gives operating systems enormous flexibility to efficiently manage physical memory

2.1.3 Timer interrupts

Process isolation also requires hardware to provide a way for the operating system kernel to periodically regain control of the processor. The implementation of a hardware timer is an answer to this.

A hardware device can cause a processor interrupt after some given delay, either in time-metric or in number of instructions executed. The interrupt frequency is set by the kernel. Note that a higher more frequent interrupts lead to a larger overhead.

2.3 Mode transfer

Mode switch – from user to kernel mode

- Interrupts
  - Triggered by timer and I/O devices
- Exceptions
  - Triggered by unexpected program behaviour (division by zero)
  - Or malicious behaviour
- System calls (aka protected procedure call)
  - Request by program for kernel to do some operation on its behalf
  - Only limited # f very carefully coded entry points

Usually switches back to user mode when kernel is done. Done by resetting the flag in the CPU

Some examples of exceptions

- User process that tries to execute a privileged instruction
- Illegal memory reference
- Illegal instruction trying to be executed by corrupted programs

Mode switch – from kernel mode to user mode

- New process/new thread start
  - Jump to first instruction in program/thread
- Return from interrupt, exception or system call
  - Resume suspended execution
- Process/thread switch
  - Resume some other process
- User-level upcall (UNIX signal)
  - Asynchronous notification to user program

Op link om booting, virtual memory og litt annet.

https://tldp.org/HOWTO/Unix-and-Internet-Fundamentals-HOWTO/anatomy.html

The operating system keeps track of the various processes on the computer using a data structure called the process control block. The process control block stores all the information the operating system needs about a particular process: block where it is stored in memory, where its executable image is on disk, which user asked it to start executing, what privileges the process has, and so forth.

2.3 Types of mode transfer

2.3.1 User to kernel mode

Interrupts

An interrupt is an asynchronous signal to the processor that some external event has occurred that may require its attention. An interrupt operates much as an exception does: the processor stops the currently executing process and starts running in the kernel at a specially designated interrupt handler. Each different type of interrupt requires its own handler. Interrupts are also used to inform the kernel of the completion of input/output requests (work is done).

An alternative to interrupts are **polling**. In a polling, the kernel loops and checks input/output device to see if an event has occurred which requires handling. If the kernel is polling, user-level code cannot be run.

Polling: kernel waits until I/O is done (overhead is very high) whereas **interrupts** gives the kernel he opportunity to work In the meantime.

In addition to sending interrupts [Devices] possibly also needs to be able to send and retrieve data.

- Device access memory
    - Programmed I/O – kind of similar to polling, CPU intensive, CPU is reading/writing to device directly (for larger amount of data this is bad)
    - Direct memory access DMA, takes CPU out of transfer equation, often handled in hardware, ex a buffer descriptor

-Os kernel needs to communicate with physical devices, hardware

- Devices operate asynchronously from the CPU

How does device interrupts work? (section 2.4)

blocking (should not for example send system calls, waiting for answer)

Processor exceptions

A processor exception is any unexpected condition caused by user program behaviour. On an exception, the hardware will stop the currently executing process and start running the kernel at a specially designated exception handler. A processor exception will be triggered whenever a process attempts to perform a privileged instruction or accesses memory outside of its own memory region. Exceptions can also be triggered by a number of benign program events. Examples are when a process divides an integer by zero, attempts to write to read-only memory etc.

System calls

A system call is any procedure provided by the kernel that can be called from user-level. A user process transitions into the operating system kernel by requesting the kernel do some operation on it's behalf. Most processors implement system calls using a **trap** instruction (but it is not required). A trap instruction, as with an interrupt or exception, changes the processors mode from user to kernel and starts executing in the kernel at a pre-defined handler. A trap instruction is caused voluntarily. To protect the kernel from misbehaving user programs, it is essential that applications trap only to a pre-defined address, they can not be allowed to jump to arbitrary places in the kernel.

Operating systems provide a substantial number of system calls. Examples include ones to establish a connection to a web server, to send or receive packets over the network, to create or delete files, to read or write data into files, and to create a new user process. To the user program, these are called just like normal procedures, with parameters and return values. Like any good abstraction, the caller only needs to be concerned with the interface; they do not need to know that the routine is actually being implemented by the kernel rather than by a library. The kernel handles all of the details of checking and copying arguments, performing the operation, and copying any return values back into the process's memory. When the kernel is done with the system call, it resumes user-level execution at the instruction immediately after the trap.

Trap: process-initiated system call

2.3.2 Kernel to user mode

Just as there are several different causes for transitions from user-mode to kernelmode, there are also several causes for transitions from kernel-mode to usermode:

Resume after an exception, interrupt or system call

When the kernel finishes handling the request, it resumes execution of the interrupted process by restoring its program counter, restoring its registers, and changing the mode back to user-level.

Switch to a different process

In some cases, such as on a timer interrupt, the kernel will decide to switch to running a different process than the one that had been running before the interrupt, exception, or system call. Since the kernel will eventually want to resume the old process, the kernel needs to save the process's state — its program counter, registers, and so forth — in the process's control block. The kernel can then resume a different process, by loading its state — its program counter, registers, and so forth — from the process's control block into the processor, and then switching to user-mode.

User-level upcall

Many operating systems provide user programs the ability to receive asynchronous notification of events. The mechanism, which we will describe shortly, is very similar to kernel interrupt handling, except at user-level

2.4 Implementing safe mode transfer

Whether transitioning from user to kernel mode, or in the opposite direction, care must be taken to ensure that a buggy or malicious user program cannot corrupt the kernel. Most os' have a common sequence of instructions for entering the kernel, at a minimum this common sequence must provide:

- **Limited entry**
  To transfer control to the operating system kernel, the hardware must ensure that the entry point into the kernel is one set up by the kernel. User programs cannot be allowed to jump to arbitrary locations in the kernel. For example, the kernel code for handling a system call to read a file will first check whether the user program has permission to do so, and if not, the kernel should return an error. Without limited entry points into the kernel, a malicious program could simply jump immediately after the code to perform the check, allowing any user access to any file in the file system. An interrupt vector table containing all the different handlers is set by the os in boot (in kernel memory).

- **Atomic changes to processor state (**atomic transfer of control, it has to happen uninterruptible**)**
  In user mode, the program counter and stack point to memory locations in the user process; memory protection prevents the user process from accessing any memory outside of its region. In kernel mode, the program counter and stack point to memory locations in the

kernel; memory protection is changed to allow the kernel to access both its own data and that of the user process. Transitioning between the two must be done atomically, so that the mode, program counter, stack, and memory protection are all changed at the same time

- **Transparent, restartable execution (**user program does not know an interrupt occurred**)**
  A running user-level process may be interrupted at any point, between any instruction and the next one. For example, the processor could have calculated a memory address, loaded it into a register, and be about to store a value to that address. The key to making interrupts work is that the operating system must be able to restore the state of the user program exactly as it was before the interrupt occurred. To the user process, an interrupt is invisible, except that the program temporarily slows down. A "hello world" program does not need to be written to understand interrupts! But an interrupt might still occur while it is running. Thus, on an interrupt, the processor saves its current state to memory, temporarily defers further events, and sets the processor to execute in kernel-mode, before jumping to the interrupt or exception handler. When the handler completes, the steps are reversed: the processor state is restored from its saved location, with the interrupted program none the wiser.

2.4.1 The interrupt vector

When an interrupt, exception or system call trap occurs, the operating system must take different actions depending on what the event which triggered the interruption. The processor will include a special register that points to an area of kernel memory called **interrupt vector**. The interrupt vector is an array of pointers, with each entry pointing to the first instruction of a different handler procedure in the kernel. The interrupt vector is stored in protected kernel memory.

When an interrupt, trap, or exception occurs, the hardware determines which hardware device caused the interrupt, whether the trap instruction was executed, or what exception condition occurred. Thus, the hardware can select the right entry from the interrupt vector and invoke the appropriate **interrupt handler.**

2.4.2 Interrupt stack

When an interrupt, processor exception, or system call trap causes a context switch into the kernel, the hardware changes the stack pointer to point to the base of the kernel's interrupt stack. The hardware automatically saves some of the interrupted process's registers by pushing them onto the interrupt stack before calling the kernel's handler.

2.4.4 Interrupt masking

The hardware provides a privileged instruction to temporarily disable or mask interrupts until it is safe to do so. This prevents for example an interrupt of being interrupted itself,

Check eax, ebx, esp registers

2.7 Starting a new process

In order for the kernel to start a new process the kernel begin with allocating and initializing the PCB. Memory is then allocated for the process before the program is copied from disk. The program counter is then set to be the first instruction of the process. Stack pointer is also set to be the base of the user stack. Finally the processor exits the kernel in the same way you would by returning a system call in order to transfer control to user mode.

2.8 Implementing upcalls

3.1.3 Process Control Block

For each process there is a Process Control Block, PCB, which stores the following (types of) process-specific information, as illustrated in Figure 3.1. (Specific details may vary from system to system.)

- Process State - Running, waiting, etc., as discussed above.
- Process ID, and parent process ID.

- CPU registers and Program Counter - These need to be saved and restored when swapping processes in and out of the CPU.
- CPU-Scheduling information - Such as priority information and pointers to scheduling queues.
- Memory-Management information - E.g. page tables or segment tables.
- Accounting information - user and kernel CPU time consumed, account numbers, limits, etc.
- I/O Status information - Devices allocated, open file tables, etc.

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/

3. The programming interface

3.1 Process management

In early batch processing systems, the kernel was in control by necessity. Now however, is to allow user programs to create and manage their own processes. An early motivation for user-level process management was to allow developers to write their own shell command line interpreters. A shell is a job control system. It allows programmers to create and manage a set of programs to do some task. Many tasks involve a sequence of steps to do something, each of which can be its own program.

3.1.1 Windows process management

In windows there is a routine called **CreateProcess** which is a system call that creates a process, that again runs a program. We call the process creator the *parent* and the process being created the *child*. The steps CreateProcess take are (simplified):

- Create and initialize the process control block (PCB) in the kernel
- Create and initialize a new address space
- Load the program prog into the address space
- Copy arguments args into memory in the address space
- Initialize the hardware context to start execution at “start”
- Inform the scheduler that the new process is ready to run

```
// simplified version of the call to create a process with arguments on Windows
simplified: boolean CreateProcess(char *prog, char *args)


// Start the child process
    if( !CreateProcess( NULL,   // No module name (use command line)
        argv[1],        // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    )
```

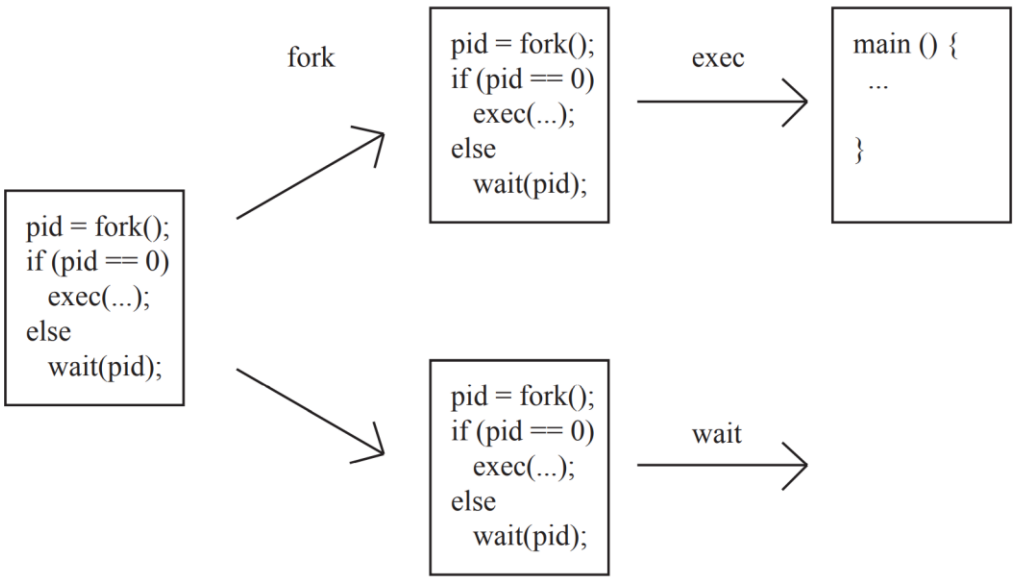
There are quite a few aspects of the process that the parent might like to control, such as: its privileges, where it sends its input and output, what it should store its files, what to use as a scheduling priority, and so forth. The real interface is more complicated than this.

3.1.2 Unix process management

Unix take somewhat of a different approach to process management. Unix split CreateProcess into two different steps.

Unix **Fork** creates a complete copy of the parent (running) process. This copy can then do whatever is necessary to set up the context of the child. It sets up privileges and priorities before calling Unix Exec. Unix fork command also takes in zero arguments

Unix **Exec** system call to change the program being run by the current process.



(if pid == 0 we know we are the child process. Fork then copies process and child calls exec)

```
int child_pid = fork();
if (child_pid == 0) {           // I'm the child process
        // getpid is a system call to get the current process's ID
        printf("I am process #%d\n", getpid());
        return 0;
} else {                                    // I'm the parent
        printf("I am parent of process #%d\n", child_pid);
        return 0;
}
```

```
Possible output:
I am parent of process 495
I am process 495

Another less likely but still possible output:
I am process 456
I am parent of process 456
```

The steps for implementing **UNIX fork** in the kernel are:

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

3.2 Input / Output

Unix io

- **Uniformity**
  All device I/O, file operations, and interprocess communication use the same set of system calls: open, close, read and write.

- **Open before use**
  Before an application does I/O, it must first call open on the device, file, or communication channel. This gives the operating system a chance to check access permissions and to set up any internal bookkeeping, does the device exist etc.

- **Byte-oriented**
  Any reads and writes returns series of bytes. All devices, even those that transfer fixed-size blocks of data, are accessed with byte arrays. Similarly, file and communication channel access is in terms of bytes, even though we store data structures in files and send data structures across channels

- **Kernel-buffered reads**
  Stream data, such as from the network or keyboard, is stored in a kernel buffer and returned to the application on request. This allows the UNIX system call read interface to be the same for devices with streaming reads as those with block reads, such as disks and Flash memory. In both cases, if no data is available to be returned immediately, the read call blocks until it arrives, potentially giving up the processor to some other task with work to do.

- **Kernel-buffered writes**
  Likewise, outgoing data is stored in a kernel buffer for transmission when the device becomes available. In the normal case, the system call write copies the data into the kernel buffer andreturns immediately. This decouples the application from the device, allowing each to go at their own speed. If the application generates data faster than the device can receive it (as is common when spooling data to a printer), the write system call blocks in the kernel until there is enough room to store the new data in the buffer.

- **Explicit close**
  When an application is done with the device or file, it calls close. This signals to the operating system that it can decrement the reference-count on the device, and garbage collect any unused kernel data structures.

3.3 Implementing a shell

The shell reads a command line from the input, and it forks a process to execute that command. UNIX fork automatically duplicates all open file descriptors in the parent, incrementing the kernel's reference counts for those descriptors, so the input and output of the child is the same as the parent. The parent waits for the child to finish before it reads the next command to execute.

- A program can be a file of commands
- A program can send/read its output to a file
- The output of a program can be the input to another program

```
main() {
        char *prog = NULL;
        char **args = NULL;

        // read the next line from the input, and parse it into the program name and its arguments
        // return false if we've reached the end of the input
        while(readAndParseCmdLine(&prog, &args)) {

                int child_pid = fork();          // create a child process to run the command

                if (child_pid == 0) {            // I'm the child process
                                                         // child uses the parent's input and output
                        exec(prog, args);        // run the program
                        // NOT REACHED
                } else {                                 // I'm the parent
                        wait(child_pid);                 // wait for the child to complete
                        return 0;
                }
        }
}
```

4. Concurrency and Threads

Operating systems often need to be able to handle multiple things at the same time. To simulate or interact with the real world, to manage hardware resources, and to map parallel applications to parallel hardware, computer systems must provide programmers with abstractions for expressing and managing concurrency.

4.1 Threads: Abstraction and interface

A thread is a single execution sequence that represents a separately schedulable task.

Threads virtualize the processor, providing the illusion that programs run on machines with an infinite number of virtual processors. The programs can then create however many threads they need without worrying about the exact number of physical processors, and each thread runs on its own virtual processor. It is the operative systems job to multiplex all of the system's threads on the actual physical processor present in the system.

A programmer uses threads to address primarily three different issues:

- **Program structure: Expressing logically concurrent taks.** Programs often interact with or simulate real world applications that have concurrent activities. Threads allow the programmer to express an application's concurrency by writing each concurrent task as a separate thread.

- **Performance: Exploiting multiple processors.** If programs can use multiple processors to do their processing in parallel, they can run faster— doing the same work in less time or doing more work in the same time. An important property of the threads abstraction is that the number of threads used by the program, does not need to match the amount of actual processors in the hardware on which it is running. The operating system transparently switches which threads are running on which processors

- **Performance: Coping with high latency I/O devices.** To do useful work, computers must interact with the outside world via Input/Output (I/O) devices. By running tasks as separate threads, when one task is waiting for I/O, the processor can make progress on a different task. We do not want the processor to be idle while waiting for I/O to complete for two reasons.

  - First, processors are often much faster than the I/O systems with which they interact, so it is useful for a programmer to multiplex the processor among multiple tasks. For example, the latency to read from disk can be tens of milliseconds, so after requesting a block from disk, an operating system switches to doing other work until the disk has copied that block to the machine's main memory.

  - Second, I/O provides a way for the computer to interact with external entities like users pressing keys on a keyboard or a remote computer sending network packets. The arrival of this type of I/O event is unpredictable, so we want the processor to be able to work on other things while still responding quickly to these events.

4.1.1 Threads: Definitions and properties

A thread is a single execution sequence that represents a separately schedulable task.

- **Single execution sequence.** By single execution sequence we mean that each thread executes a sequence of instructions just as in the familiar programming model

- **Separately schedulable task.** By separately schedulable task we mean that the operating system can run, suspend, or resume a thread at any time.

  Threads provide the illusion of an infinite number of processors. To map an arbitrary set of threads to a fixed set of processors, operative systems include a scheduler that can switch between threads that are currently running and threads that are ready to run. Switching between threads is transparent to code being executed by the threads. Threads thus provide an execution model in which each thread runs on a *dedicated virtual processor with unpredictable and variable speed.*

4.2 Simple API and example

- Thread_create(thread, func, args)
  - Creates a new thread to run func(args)
- Thread_yield()
  - Giving up processor voluntarily to schedule another thread
- Thread_join(thread)
  - In parent, wait for forked thread to exit, then return
- Thread_exit()

- Quit thread and clean up, wakes up joiner if any

4.3 Thread internals

4.3.1 Thread control block (TCB) and per-thread state

The data structure that hold the thread's state is called the thread control block (TCB). For every thread the operating system creates, it will create one TCB. The TCB holds two types of information

- The state of the computation being performed by the thread
- Metadata about the thread that is used to manage the thread

Per-thread computation state

Each thread represents its own sequentially executed computation, so to create multiple threads, we must allocate per-thread state to represent the current state of each thread's computation. This is done by adding a *stack* and a *copy of processor registers*.

Stack: A thread's stack is the same as the stack for a single-threaded computation— it stores information needed by the nested procedures the thread is currently running. A pointer for the thread's stack is stored in the TCB

Copy of processor registers: The TCB also contains fields in which the os can store a copy of all of the processors registers because a thread can be suspended and later resumed.

Per-thread metadata.

The TCB also includes include per-thread metadata— information about each thread. This information is used by the operating system to manage the thread. For example, each thread might have a thread ID, scheduling priority, and any other information the operating system wants to remember about the thread.

Shared state

In addition to per-thread state that is allocated for each thread, as Figure 4.8 illustrates, other state is shared by different threads in a multithreaded process or multi-threaded operating system. In particular, the program code is shared by all of the threads in a process. Additionally, statically-allocated global variables and dynamically-allocated heap variables can store information that is accessible to all threads.

4.3.2 Thread lifecycle

Init. Thread creation puts a thread into its initial Init state and allocates and initializes per-thread data structures.

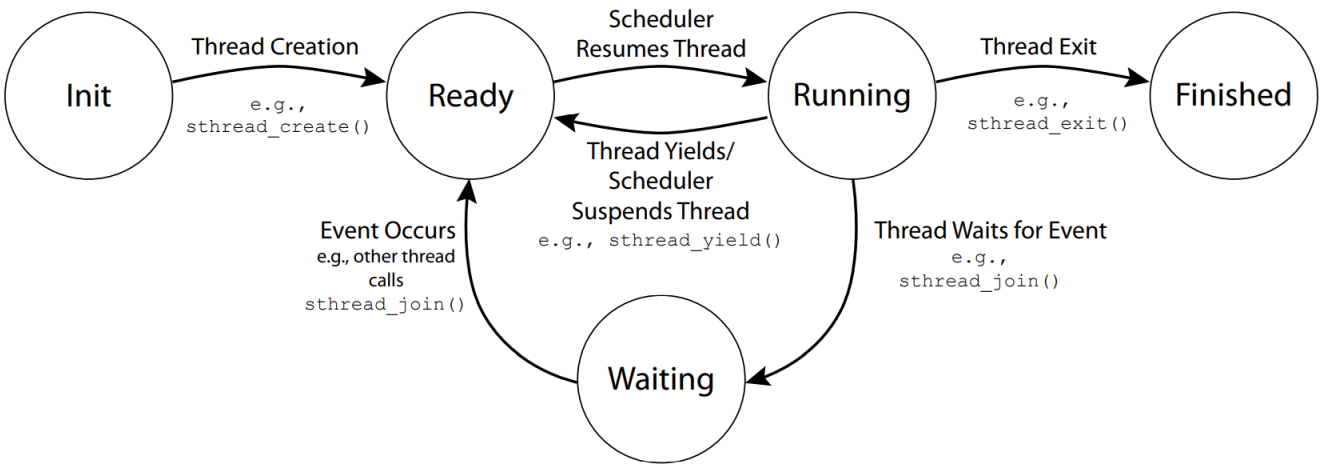
Ready. A thread in the Ready state is available to be run but it is not currently running.

Running. A thread in the Running state is running on a processor. When a thread is Running, its register values are stored on a processor rather than in the TCB data structure. A Running thread can transition to the Ready state in two ways.

- The scheduler can preempt a running thread and move it to the Ready state at any time by saving the thread's registers to its TCB and switching the processor to running the next thread on the ready list.
- A running thread can voluntarily relinquish the processor go from Running to Ready by calling yield() (e.g., sthread yield() in the sthreads library.)

Waiting. A thread in the Waiting state is waiting for some event. Whereas a thread in the Ready stage is eligible to be moved the Running state by the scheduler, a thread in the Waiting state is not eligible to be run until some action by another thread moves it from the Waiting state to the Ready state

Finished. Finally, a thread in the Finished state will never run again. The system can free some or all of its state for other uses



4.4 Implementation details

4.4.1 Creating a thread

Allocating per-thread state. To allocate a thread, we must first allocate its per-thread state. The thread constructor allocates a thread control block (TCB) and a stack. The TCB is just a data structure with fields for the state we want to be able to maintain for a thread. The stack is just an area of memory like any other data structure allocated in memory. The stack may either be fixed size or dynamic. We also have to initialize the per-thread states, by setting the TCB's registers to some values.

4.4.2 Deleting a thread

To delete a thread, we need to (1) remove it from the ready list so that it will never run again and (2) free the per-thread state we allocated for it.

A thread transitions to the Finished state by moving its TCB from the ready list to a list of finished threads the scheduler should never run. Then, any time the thread library code runs (e.g., when some other thread calls yield() or when an interrupt occurs), the thread library can call a method to free the state of any threads on the finished list.

4.4.3 Thread context switch

A *thread context switch* suspends execution of the currently running thread and resumes execution of some other thread. This is done by copying the currently running thread's registers from the processor to the thread's TCB and then copying the other thread's registers from that thread's TCB to the processor.

What causes a kernel thread context switch?

When a thread in the kernel is running, two things can cause a thread context switch.

Firstly, the currently running thread may voluntarily give up the processor and context switch the processor to some other thread that is ready to run, by running the thread_yield().

Secondly, an interrupt or exception may invoke an interrupt handler, which must save the state of the running thread, execute the handler's code, and switch to some thread that is ready to run (or it may just restore the state and resume running the thread that was just interrupted.). To prevent threads to monopolize the processor, likewise with processes a hardware timer might cause a timer interrupt.

How does a kernel thread context switch work?

1. Copy the running thread's registers from the processor to the thread's TCB

2. Copy the ready thread's registers from the thread's TCB to the processor

- The state is firstly saved, then either if it is a hardware triggered switch, the kernel's handler is ran/ and if it is a library called switch (thread.yield()) the code that handles the library call is ran. Finished by restoring the next ready threads registers.

4.4.4 Multi-threaded processes

- **Multi-threaded kernel.** Multiple threads, sharing kernel data structures, capable of using privileged instructions

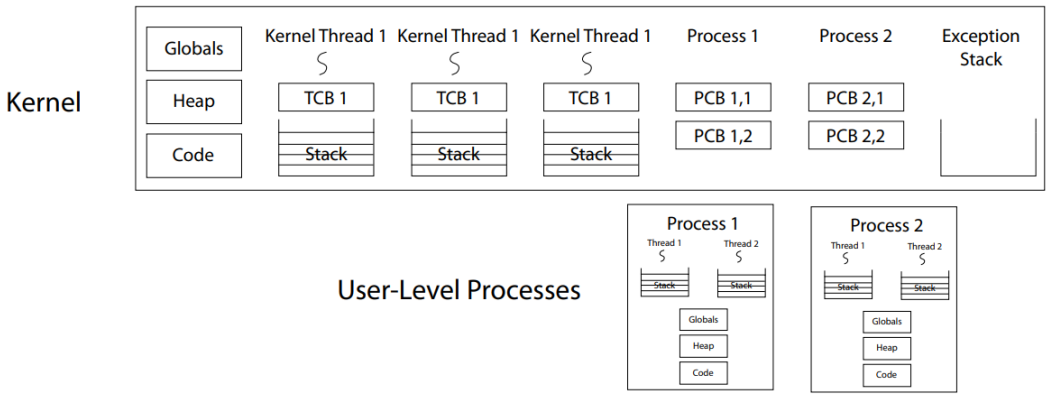


Figure 4.15: A multi-threaded kernel with 3 kernel threads and 2 user-level processes, each with 2 threads.

- **Multiprocess kernel.** Multiple single-threaded user processes, uses system calls to access shared kernel data structures
- **Mutliple multi-threaded user processes.** Each with multiple threads, sharing same data structures, isolated from other user processes via memory protection etc.

Two different implementations

- **Pre-emptive:** interrupts, save ones thread and restore the state of another (involuntarily, interrupt received by the CPU)
- **erative:** Yield, save ones thread's state and restore another (Voluntarily)

4.5 Multi-threaded programs – advantages / disadvantages

Multi-threaded program advantages:

- Less overhead to establish and terminate vs. a process: because very little memory copying is required (just the thread stack), threads are faster to start than processes. To start a

process, the whole process area must be duplicated for the new process copy to start. While some operating systems only copy memory once it is modified (copy-on-write), this is not universally guaranteed.

- Faster task-switching: in many cases, it is faster for an operating system to switch between threads for the active CPU task than it is to switch between different processes. The CPU caches and program context can be maintained between threads in a process, rather than being reloaded as in the case of switching a CPU to a different process.
- Data sharing with other threads in a process: for tasks that require sharing large amounts of data, the fact that threads all share a process's memory pool is very beneficial. Not having separate copies means that different threads can read and modify a shared pool of memory easily. While data sharing is possible with separate processes through shared memory and inter-process communication, this sharing is of an arms-length nature and is not inherently built into the process model.
- Threads are a useful choice when you have a workload that consists of lightweight tasks (in terms of processing effort or memory size) that come in, for example with a web server servicing page requests. There, each request is small in scope and in memory usage. Threads are also useful in situations where multi-part information is being processed – for example, separating a multi-page TIFF image into separate TIFF files for separate pages. In that situation, being able to load the TIFF into memory once and have multiple threads access the same memory buffer leads to performance benefits.

Multi-threaded program disadvantages:

- Synchronization overhead of shared data: shared data that is modified requires special handling in the form of locks, mutexes and other primitives to ensure that data is not being read while written, nor written by multiple threads at the same time.
- Shared process memory space: all threads in a process share the same memory space. If something goes wrong in one thread and causes data corruption or an access violation, then this affects and corrupts all the threads in that process. This is a special concern for cross-language environments where it is very easy to have subtle ABI interaction problems, such as Java-based web servers calling upon native libraries via the JNI (Java Native Interface) ABI.

- Program debugging: multi-threaded programs present difficulties in finding and resolving bugs over and beyond the normal difficulties of debugging programs. Synchronization issues, non-deterministic timing and accidental data corruption all conspire to make debugging multi-threaded programs an order of magnitude more difficult than single-threaded programs.

5. Synchronizing access to shared objects

Sharing state among threads is useful because it allows threads to communicate, to coordinate work, and to share information. However, when cooperating threads use shared state, writing correct multi-threaded programs becomes much more difficult. *Cooperative threads* are proned to mainly three different obstacles.

1. **Program execution depends on the interleaving's of the thread's access to shared state**
   For example, if two threads write a shared variable, the final value of the the variable depends on which of the threads' writes finishes last.

2. **Program execution can be nondeterministic**
   Different runs of the same program may produce different results. For example, the scheduler may make different scheduling decisions, the processor may run at a different frequency, or another concurrently running program may affect the cache hit rate.

3. **Compilers and architectures reorder instructions**
   Modern compilers and hardware will reorder instructions to improve performance. This reordering is generally invisible to single-threaded programs; compilers and processors take care to ensure that dependencies within a sequence of instruction are preserved. However, this reordering can become visible when multiple threads interact and observe intermediate states.

Structured synchronization. Given all of these challenges, multi-threaded code can introduce subtle, non-deterministic, and unreproducible bugs. This chapter describes a structured synchronization approach to sharing state in multi-threaded programs. Rather than scattering access to shared state throughout the program and attempting ad hoc reasoning about what happens when the threads' accesses are interleaved in various ways, we structure the program to facilitate reasoning about it and we use a set of standard synchronization primitives to control access to shared state.

5.1 Challenges

Why are unstructured multi-threaded programs difficult to reason about?

5.1.1 Race conditions

A race *condition* is when the behaviour of a program depends on the interleaving of operations of different threads. In effect, the threads run a race between their operations, and the results of the program execution depends on who wins the race.

5.1.2 Atomic operations

An atomic operation is an indivisible operation that cannot be interleaved or split with or by other operations. On most modern architectures a load or store of a 32-bit word from or to memory is an atomic operation. Conversely, a load or store is not always an atomic operation. Depending on the implementation of the hardware, if two threads store the value of a a 64-bit floating point register to a memory address, the final result might be the first value, the second value, or a mix of the two.

5.1.3 Too much milk

Roommate 1's actions	Roommate 2's actions
3:00 Look in fridge; out of milk	
3:05 Leave for store	
3:10 Arrive at store	Look in fridge; out of milk
3:15 Buy milk	Leave for store
3:20 Arrive home; put milk away	Arrive at store
3:25	Buy milk
3:30	Arrive home; put milk away
3:35	Oh no!

Non-atomic operations may be interrupted by other threads.

An example to fix this would be by using *mutual exclusion*. Only one thread does a particular thing at a time. The *Critical section* would be the piece of code that only one thread can execute at once.

We could also want to *lock* the critical section. Preventing someone from doing something until we unlock it after we are finished. Lock before accessing shared memory.

See Peterson's algorithm for generalizing thread synchronization.

5.2 Shared objects and synchronization variables

A multi-threaded program is built using *shared objecs*. Shared objects are objects that can safely be accessed by multiple threads. All shared state in a program—including variables allocated on the heap (e.g., objects allocated with malloc() or new()) and static, global variables—should be encapsulated in one or more shared objects. The shared objects hide the details of synchronizing the actions of multiple threads behind a clean interface.

Implementing shared objects

- **Shared objects**
  As in standard object-oriented programming, the shared objects define the application-specific logic and hide the internal implementation details. Externally, shared objects appear essentially the same as objects you define for single-threaded programs.

- **Synchronization variables**
  Shared objects include synchronization variables as member variables. Synchronization variables are instances of carefully designed classes that Synchronization variables provide broadly-useful primitives for synchronization. In particular, we build shared objects using two types of synchronization variables: *locks* and *condition variables*

  Synchronization variables coordinate access to state variables, which are just the normal member variables of an object that you are familiar with from single-threaded programming(ints, strings etc)

- **Atomic read-modify-write instructions**
  Atomic read-modify-write instructions atomic read-modify-write instruction allows one thread to have exclusive access of a memory location while the instruction's read, modification, and write of that location are guaranteed not to be interleaved with any other thread's access to that memory.

5.3 Lock: Mutual exclusion

A *lock* is a synchronization variable that provides mutual exclusion—when one thread holds a lock, no other thread can hold the same lock. A program associates each lock with some subset of shared state and requires a thread to hold the lock when accessing that state. Then, only one thread can be accessing the shared state at a time. In the point of view of other threads, the thread performing some arbitrary operation (holding the lock), appears to be atomic. They cannot access shared state, because they are not holding the lock.

A lock can either be in the state *busy* or *free.* Lock :: acquire() and Lock :: release(). Acquire will check is state is busy or free, and it is all an atomic operation so no other threads can acquire it.

A lock should ensure the following three properties:

1. **Mutual Exclusion**. At most one thread holds the lock.
2. **Progress**. If no thread holds the lock and any thread attempts to acquire the lock, then eventually some thread succeeds in acquiring the lock.
3. **Bounded waiting**. If thread T attempts to acquire a lock, then there exists a bound on the number times other threads successfully acquire the lock before T does.

As in standard object oriented programming, each shared object is an instance of a class that defines the class's state and the methods that operate on that state. When we use a class's constructor o instantiate a shared object we allocate both a new lock and new instances of the state protected by that lock.

5.4 Condition variables: Waiting for a change

A condition variable is a synchronization object that enables a thread to effi ciently wait for a change to shared state that is protected by a lock.

Condition variables provide a way for one thread to wait for another thread to take some action. For example a web server might want to wait until a new request arrives instead of returning an error. Broadly speaking, what we want to do is have a thread wait for something else to change the state of the system so that it can make progress.

A condition variable has three methods:

- **CV :: wait(Lock *lock)**. Atomically releases the lock and suspends execution of the calling thread, placing the calling thread on the condition variable's waiting queue. Later, when the calling thread is reenabled, it reacquires the lock before returning from the wait() call.

- **CV :: signal().** Takes one waiting thread off the condition variable's waiting queue and marks it as eligible to run (i.e., it puts the thread on the scheduler's ready list.)

- **CV :: broadcast().** Takes all waiting threads off the condition variable's waiting queue and marks them as eligible to run.

Condition variables integration with locks

- **A condition variable is memoryless; the condition variable, itself, has no internal state other than a queue of waiting threads.**
  Condition variables do not need state of their own because they are always used within shared objects that define their own state.

- **Wait() atomically releases the lock.**
  The atomicity ensures that there is no separation between checking the shared object's state, deciding to wait, adding the waiting thread to the condition variable's queue, and releasing the lock so that the other thread can access the shared object to change its state and signal.

- **When a waiting thread is reenabled via signal() or broadcast(), it may not run immediately.**
  When a waiting thread is re-enabled, it is moved to the scheduler's ready queue with no special priority, and the scheduler may run it at some later time. This is needed for other concurrent programs to finish as well.

5.4

5.4.2 Thread life cycle revisited

A Running thread that calls Cond::wait() is put in the Waiting state. This is typically implemented by moving the thread's thread control block (TCB) from the scheduler's ready queue to a queue of waiting threads associated with that condition variable. Later, when some other Running thread calls Cond::signal() or Cond::broadcast() on that condition variable, one (if signal() or all (if broadcast() of the TCBs on that condition variable's waiting queue are moved to the scheduler's ready list. This changes those threads from the Waiting state to the Ready state. At some later time, the scheduler selects a Ready thread and runs it by moving it to the Running state.

Locks are similar. Lock::acquire() on a busy lock puts caller into the Waiting state, with the caller's TCB on a list of waiting TCBs associated with the lock. Sometime later, when the lock owner calls Lock::release(), the TCB is moved to the ready list and the thread transitions to the Ready state.

Notice that threads that are Running or Ready have their state located at a pre-defined, "global" location like the CPU (for a Running thread) or the scheduler's list of ready threads (for a Ready thread). However, threads that are Waiting typically have their state located on some per-lock or per-conditionvariable queue of waiting threads. Then, a Cond::signal(), Cond::broadcast(), or

Lock::release() call can easily find and reenable a waiting thread for that particular condition variable or lock.

Semaphores

Locks vs semaphores

First, using separate lock and condition variable classes makes code more self-documenting and easier to read. As the quote from Dijkstra above notes, there really are two abstractions here, and code is clearer when the role of each synchronization variable is made clear through explicit typing.

Second, a stateless condition variable bound to a lock turns out to be a better abstraction for generalized waiting than a semaphore. By binding a condition variable to a lock, we can conveniently wait on any arbitrary predicate on an object's state. In contrast, semaphores rely on carefully mapping the object's state to the semaphore's value so that a decision to wait or proceed in P() can be made entirely based on the value, without holding a lock or examining the rest of the shared object's state.

There is one situation in which semaphores are often superior to condition variables and locks: synchronizing communication between an I/O device and threads running on the processor

5.5 Implementing synchronization objects

5.5.1 Spinlocks

A spinlock is a lock where a processor/thread waits in a loop for the lock to become free. This approach will be inefficient if locks are held during long operations on shared data, but if locks are known to only be held for short periods (i.e., less time than a context switch would take), spinlocks make sense. So, spinlocks are frequently used in multiprocessor kernels for shared objects whose methods all run fast.

6. Advanced Synchronisation

6.1 Multi/object synchronisation

A general problem that arises whenever a program contains multiple shared objects accessed by threads is that, even if each object has a lock and guarantees that its methods operate atomically, sequences of operations by different threads across different objects can be interleaved.

6.1.1 Fine-grained locking

Partition an object's state into different subsets protected by different locks.

6.1.2 Per-processor data structures

Partition object so that most/all accesses are made by threads running on the same processor.

6.1.3 Ownership pattern

A thread removes an object from a container and then may access the object without holding a lock because the program structure guarantees that at most one thread owns an object at a time.

6.1.4 Staged architecture

A staged architecture divides a system into multiple subsystems called stages, where each stage includes some state private to the stage and a set of one or more worker threads that operate on that state. Different stages communicate by sending messages to each other via shared producer-consumer queues, and each worker thread repeatedly pulls the next message from a stage's incoming queue and then processes it, possibly producing one or more messages for other stages' queues.

Synchronisation performance

A program with lots of concurrent threads can still have poor performance on a multiprocessor. There are several reason for this. One may be the overhead accumulated when creating threads if not needed. Lock contention is also a cause to this. Lock contention means that only one thread may hold a given lock. False sharing may also happen, communication between cores even for data that is not shared. And shared data protected by a lock may ping back and forth between cores.

6.2 Deadlock

A challenge to constructing programs that include multiple shared objects is deadlock. Deadlock is a cycle of waiting among a set of threads where each thread is waiting for some other thread in the cycle to take some action.

Deadlock: a state in which each member of a group is waiting for another member, including itself, to take action

The four necessary conditions for a deadlock are the following:

1. Mutual exclusion: At least one resource must be held in a non-sharable mode. Otherwise the resource would not be prevented to be used by another process.
2. No pre-emption: a resource can be released only voluntarily by the process holding it.
3. Hold and wait: A process is currently holding at least one resource and requesting addition resources which are being held by other processes.
4. Circular wait: Each process must be waiting for resources held by another process, which again is waiting for another process. Circular chain of requests

6.2.3 Preventing deadlocks

In order to prevent or avoid deadlocks, we only have to eliminate one of the conditions necessary for a deadlock to appear. There are three main methods of preventing a deadlock

- Exploit or limit program behaviour
  - Limit program from doing anything that might lead to a deadlock
- Predict the future
  - If we know what the program will do, we might tell if granting a resource might lead to a deadlock
- Detect and recover
  - If we rollback a thread, we can fix deadlocks when they occur

Exploit or limit program behaviour

Provide enough resources. We might *eliminate wait while holding*, so releasing a lock when calling is out of module. *Eliminate circular waiting* by using lock ordering. Always has to acquire lock in a fixed order.

Predict the future

Bankers algorithm :

https://www.geeksforgeeks.org/bankers-algorithm-in-operating-system-2/

Detect and recover

An operating system may actively detect a deadlock state by creating a resource allocation graph. This graph contains information about all the instances of all the resources and whether they are available or being used.

6.3 Lock contention

6.3.1 MCS lock

MCS is an implementation of a spinlock optimized for the case when there are a significant number of waiting threads.

BIIIIIIIG ARTIKKEL!

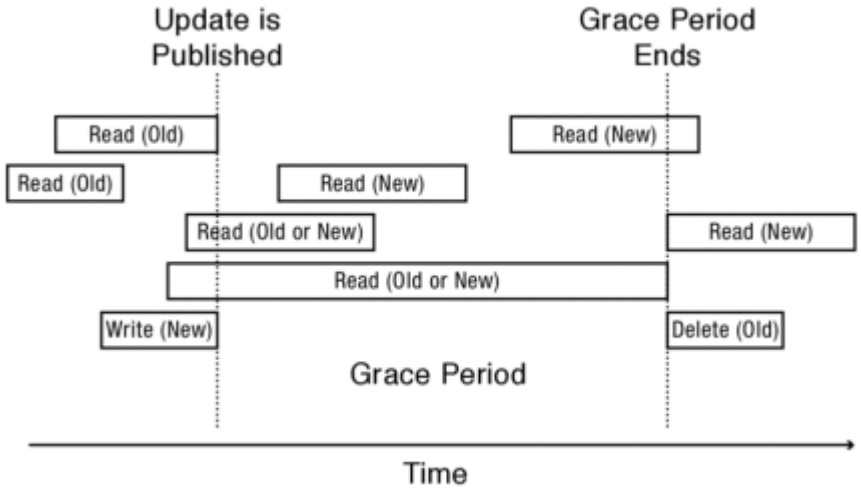
https://lwn.net/Articles/590243/

6.3.2 RCU lock (Read-Copy-Update)

RCU provides high-performance synchronization for data structures that are frequently read and occasionally updated. RCU reduces the overhead for readers at a cost of increased overhead for writers.

The RCU approach helps battle a crucial problem of the standard readers-writers locks, namely the cache behaviour for concurrent reads with short critical sections. In particular, on a multiprocessor, when one processor updates a hardware cache line, it invalidates that cache line in all other caches. Then, when another processor wants to read and update the data, it first suffers a cache miss and must fetch the data from the first processor's cache or from main memory; then it must invalidate the cache line in other caches; finally, it can update the data in its cache. This process of fetching and invalidating cache line, may take a processor hundreds of cycles.

RCU allows up to one read/write critical section to be concurrent with any number of read-only critical sections. However, writers have to guarantee that an old version is not freed until all readers have finished accessing it. We cannot update a version midst-reading. The method relies on atomic read-modify-write instructions.



Implementation

- Readers disable interrupts on entry
  - Guarantees they complete critical section in a timely fashion
  - No read or write lock
- Writer
  - Acquire write lock
  - Compute new data structure
  - Publish new version with atomic instruction
  - Release write lock
  - Wait for time splice on each CPU
  - Only then, garbage old version of data structure

7. Scheduling

Parallel applications can create more work than processors, and if care is not taken in the design of the scheduling policy, performance can badly degrade.

7.1 Uniprocessor scheduling

7.1.1 First-In-First-Out FIFO

A FIFO scheduling algorithm executes each task in the order in which it arrives. A FIFO-scheduler is very work conserving (not leaving the processor idle when there is work to do) and minimizes overhead, switching between tasks only when they are finished. Has best throughput when fixed sized tasks.

A FIFO scheduling algorithm may be optimal when tasks processed are equal in size and they are pure computational, i.e. they only need the processor to complete. This is because the FIFO algorithm minimizes overhead by only switching a task when it is complete. If however a long task would enter the scheduler first, and several smaller tasks at the end, the FIFO scheduler would result in a poor average response time.

7.1.2 Shortest Job First SJF

Utilizing a shortest job first algorithm is an optimal policy for minimizing average response time.

SJF however can suffer from starvation and frequent context switches.

7.1.3 Round Robin

In a Round Robing algorithm, the scheduler assigns the processor a timer interrupt for some delay, so called *time quantum.* At the end of each time quantum, if the task is not finished, it is pre-empted and the processor is given the next task on the ready list. This method ensures no starvation, as each task will eventually reach the front of the queue and receive its time quantum.

However, Round Robin may also suffer from a lot of overhead if time quantum's are too low. The scheduler would consequentially spend all of its time context switching, not getting a lot of work done.

May also perform poorly when running a mixture of I/O -bound and computer tasks. I/O requests are often very short. Yielding the processor even though the request is finished, leaving rest of the request to be reassigned a new time quantum sucks.

7.1.4 Max-Min Fairness

Max-min fairness iteratively maximizes the minimum share (of computing power) to each process whose demand is not fully serviced, until all resources are assigned. If a given task needs less than an equal share, schedule the smallest of these first, then split the remaining share on the next tasks in the queue.

7.1.5 Multi-Level Feedback MFQ

Aims to achieve these goals (a reasonable compromise, not perfect of course):

- Responsiveness – Run short tasks quickly
- Low Overhead – Minimize the number of pre-emption's, as in FIFO
- Starvation-Freedom – All tasks should make progress, such as in Round Robin
- Background tasks - . Defer system maintenance tasks, such as disk defragmentation, so they do not interfere with user work
- Fairness - Assign (non-background) processes approximately their max-min fair share of the processor.

MFQ is an extension of Round Robin. Instead of only a single queue, MFQ has multiple Round Robin queues, each with a different priority level and time quantum. Tasks at a higher priority level preempt lower priority tasks, while tasks at the same level are scheduled in Round Robin fashion. Further, higher priority levels have shorter time quanta than lower levels.

However, the MFQ scheduler monitors every process to ensure it is receiving its fair share of the resources, preventing starvation and enforcing max-min fairness.

7.1.6 Summary

We summarize the lessons from this section:

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.
- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.
- Max-min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

7.2 Multiprocessor scheduling

The rising number of processors in computers poses two different challenges

1. How do we make effective use of multiple cores for running sequential tasks?
2. How do we adapt scheduling algorithms for parallel applications?

7.2.1 Scheduling Sequential Applications on Multiprocessors

A multilevel feedback queue MFQ may help tackle these obstacles. A multilevel feedback queue works by assigning processes into queues when the processes enter the system. The processes have the unique ability to move between queues depending on how taxing they are on the CPU. If a process uses too much CPU time, it will be moved to a lower-priority queue and likewise for a process that waits to long in a lower-priority queue, it will be moved to a higher-priority queue. This form of aging helps prevent starvation.

However using a MFQ on a multi core processor rises a couple of challenges. Firstly, a potential performance issue on the MFQ may be a large amount of contention for the MFQ lock, especially when there are a large amount of processors. Secondly, another issue of the MFQ is the latency caused by fetching data from a remote cache. Each processor will need to fetch the current state of the MFQ from the cache of the previous processor to hold the lock. Lastly limited cache reuse is also reason for its inefficiency. If threads run on the first available processor, they are likely to be assigned to a different processor each time they are scheduled. This means that any data needed by the thread is unlikely to be cached on that processor, and therefore need to switch processor to access the correct cache.

The latter of the three problems may be resolved by using affinity scheduling. Once a thread is scheduled on a processor, it is returned to the same processor when it is re-scheduled, maximising cache reuse.

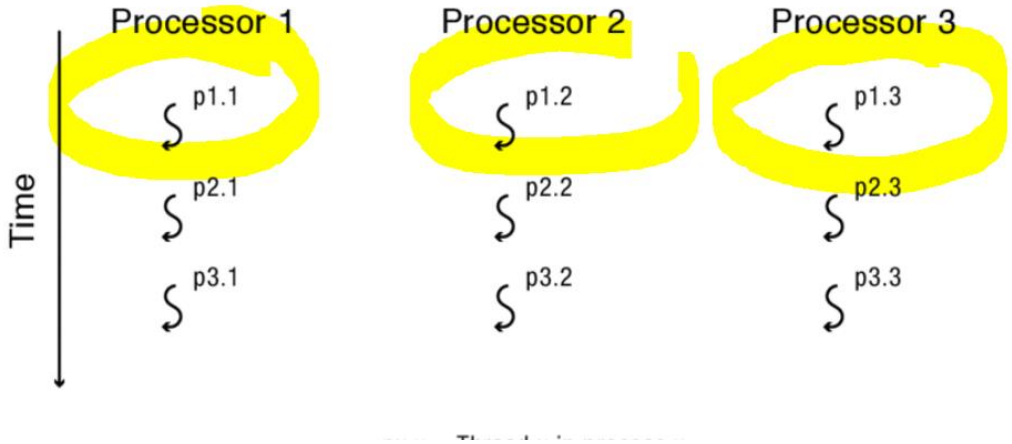
7.2.2 Scheduling Parallel Applications

For example, an image processing application may divide the image up into equal size chunks, assigning one to each processor. While the application could divide the image into many more chunks than processors, this comes at a cost in efficiency: less cache reuse and more communication to coordinate work at the boundary between each chunk.

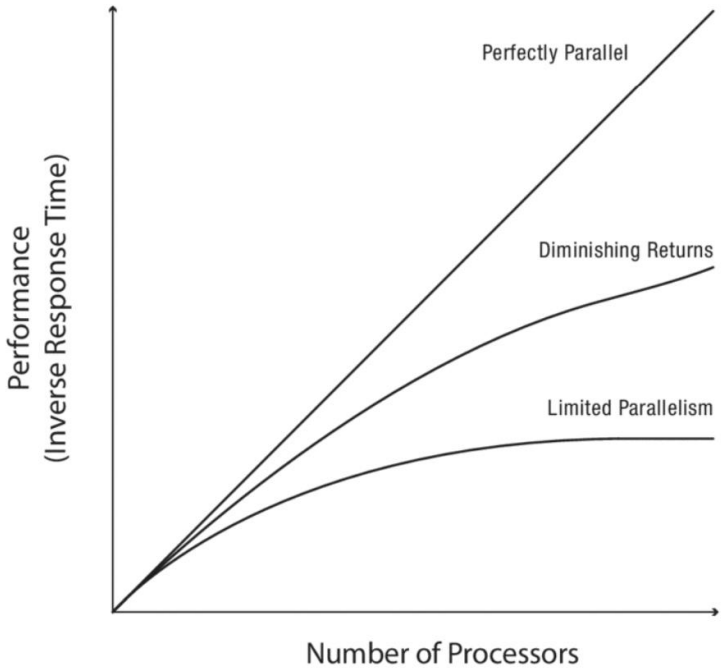
Bulk synchronous parallelism is a design pattern for parallel scheduling. It works by splitting work in to roughly equal sized chunks, once all the chunks finish, the processor synchronize at a barrier before communicating their results. This however may result in a “Bulk synchronous delay”. At each step, the computation is limited by the slowest processor to complete that step. If a processor is preempted, its work will be delayed, stalling the remaining processors until the last one is scheduled.

Oblivious scheduling: as the operating system scheduler operates without knowledge of the intent of the parallel application — each thread is scheduled as a completely independent entity

Gang scheduling: schedule all of the tasks of a program together. The os may also pre-empt all of the processors of an application in order to make space for a new different application.



An implication of the figure under, is that it is usually more efficient to run two parallel programs each with half the number of processors, than to time slice the two programs, each gang scheduled onto all of the processors. Allocating different processors to different tasks is called space sharing.



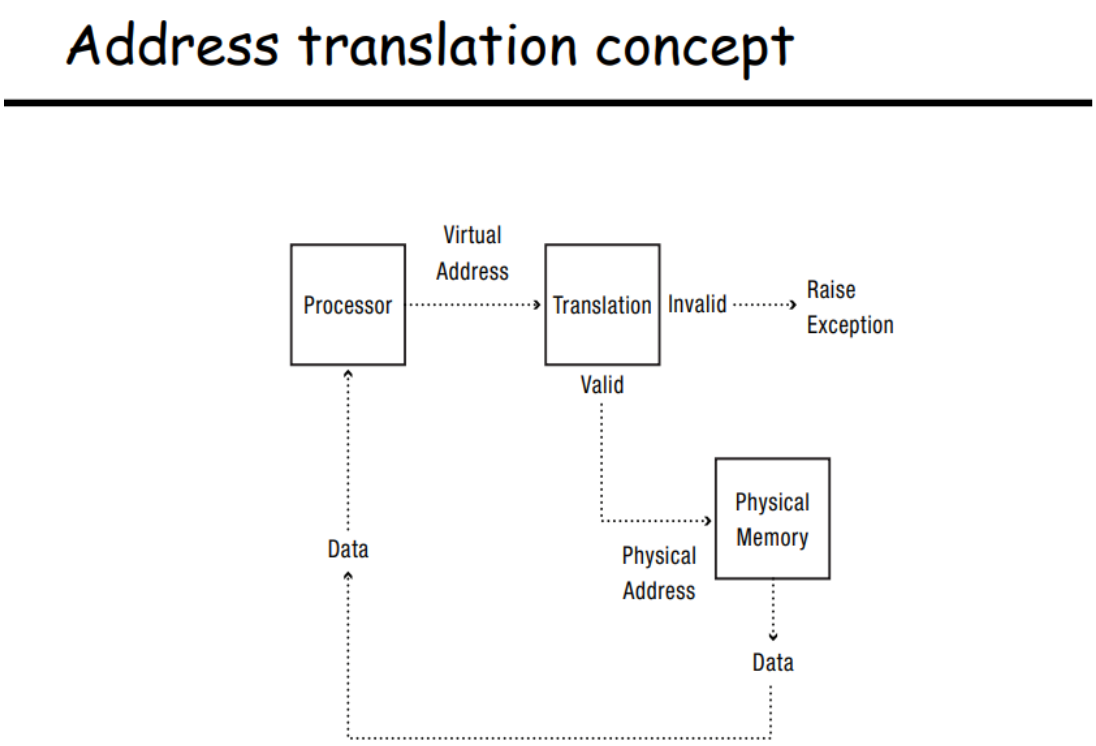
8. Address translation

Virtual Memory: the address space a process can "see". Size of address space is dependant on instruction set architecture (register size)

- 64 bit cpu, theoretical limit 64 bit memory addresses (really though, 48 bit, 40 bits)

8.1 Address translation concept

The translator converts instructions and data memory from the processor to real physical memory addresses.



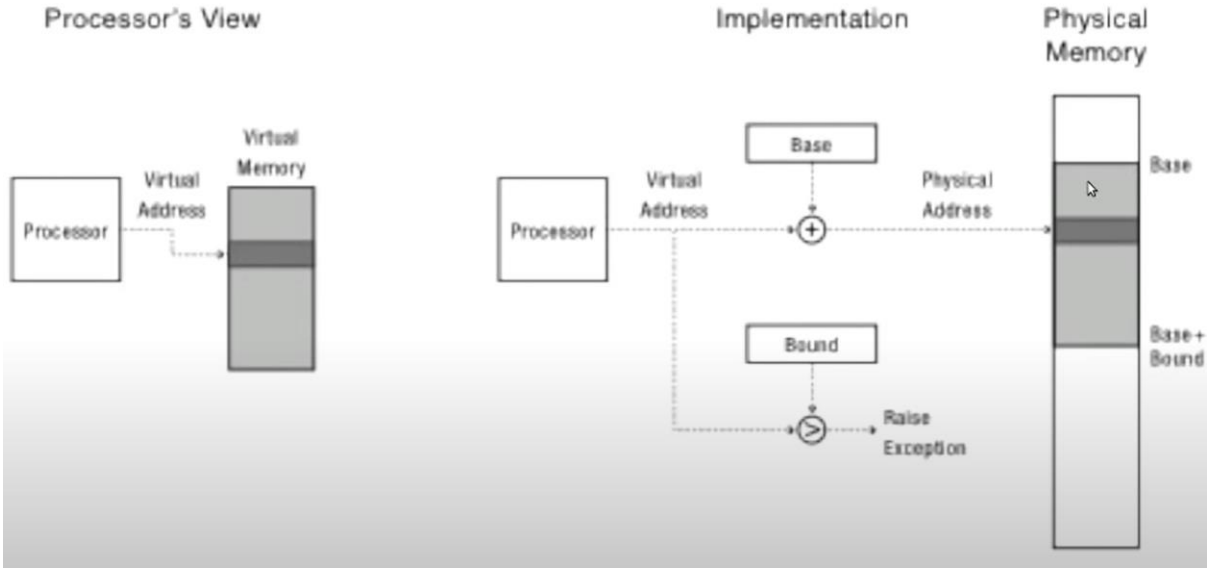
Goals

- Memory protection
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Multiple regions of dynamic allocation (heaps / stacks)
- Efficiency

- Memory placement
- Runtime lookup
- Compact translation tables

- Portability

Virtually addressed base and bounds



The base address is added to the virtual address that and points it to the base of the actual physical memory.

With virtually addressed base and bounds, what is saved/restored on a process context switch?

The base and bounds registrers that are added to the virtual address that each processes references, must be updated (saved/restored) alongside the process switch.

Pros	Cons
Simple	Cant keep program from accidentally overwriting its own code (cant specify access point on memory layout)
Fast (2 registrers, adder, comparator)	Cant share code/data with other processes, base and bounds are to restrictive
safe	Cant grow stack/heap as needed
Can relocate in physical memory without changing process (easy to update base/bounds)	

8.2 Translation Lookaside Buffer (TLB)

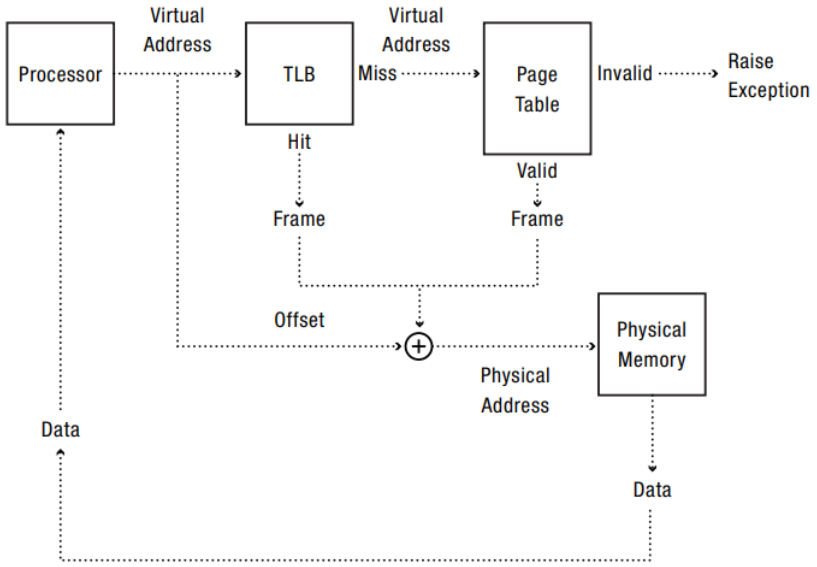
We have now studied different methods for converting virtual address spaces to physical memory. This however is a time-consuming task in a larger quantity. Introduce the idea of a cache that will hold recent translation between virtual pages and frames.

The *Translation Lookaside buffer (TLB)* is a cache that stores recent conversions or memory translation in a small, superfast memory (hardware). If we have a cache hit, meaning if we find the virtual address that we are looking to translate in the TLB, then we can just use that translation from the cache memory. If we miss, we have to walk the multi-level page table like in general.

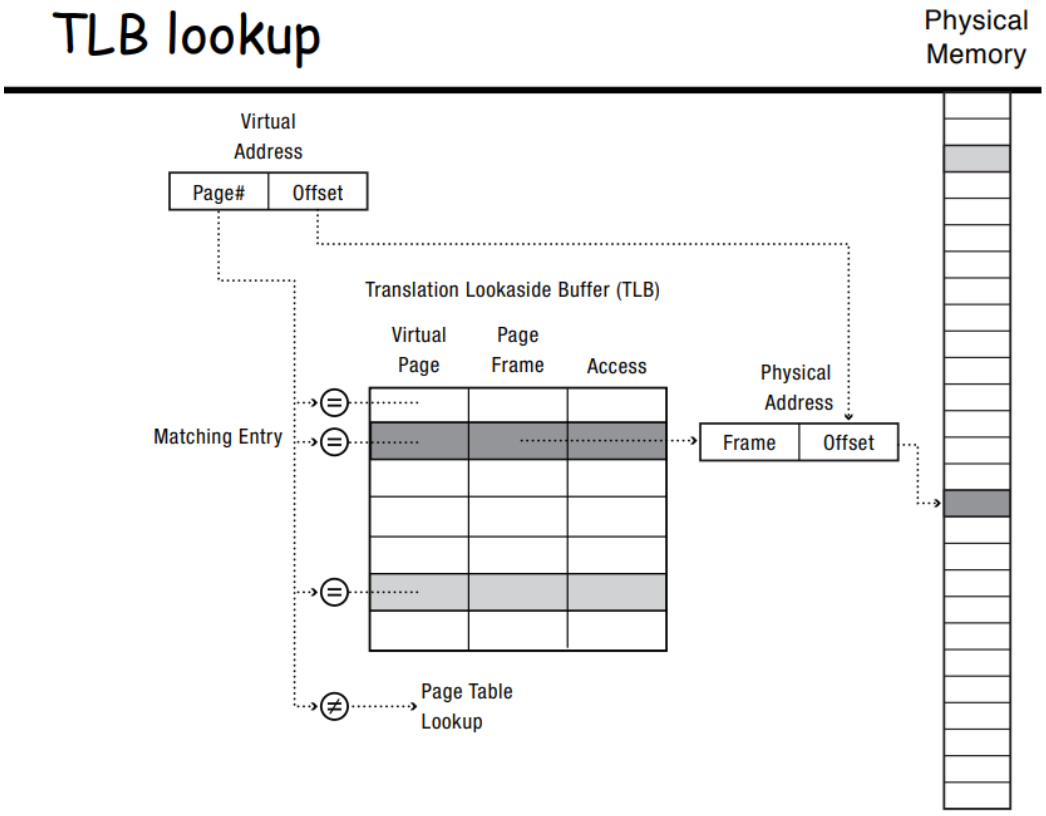
The cost of translation = the cost of TLB lookup + probability(TLB Miss) * cost of page table lookup

The TLB contains the virtual addresses of a certain process that is running. When we have a context switch, we have to flush the TLB so the new process does not have access to memory space that is ours. A solution to this is tagging the TLB. By giving the TLB a process ID, we can check if a current process matches TLB before we check for hits.

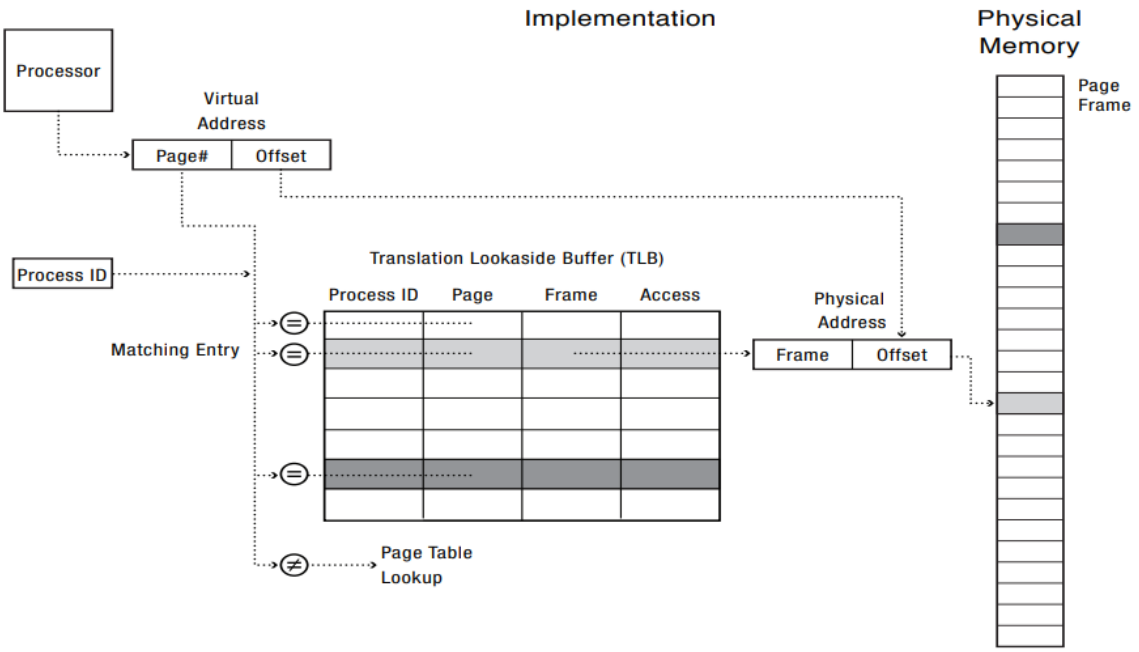
TLB and page table translation



TLB lookup



Tagged TLB



Memory Hierarchy

Cache	Hit Cost	Size
1st level cache/first level TLB	1 ns	64 KB
2nd level cache/second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

Address Translation Uses

- Process isolation
  - Keep a process from touching anyone else's memory, or the kernel's
- Efficient interprocess communication
  - Shared regions of memory between processes
- Shared code segments
  - E.g., common libraries used by many different programs
- Program initialization
  - Start running a program before it is entirely in memory
- Dynamic memory allocation
  - Allocate and initialize stack/heap pages on demand

Address Translation (more)

- Program debugging
  - Data breakpoints when address is accessed
- Zero-copy I/O
  - Directly from I/O device into/out of user memory
- Memory mapped files
  - Access file data using load/store instructions
- Demand-paged virtual memory
  - Illusion of near-infinite memory, backed by disk or memory on other machines

Flexible address translation

8.3 Segmentation

8.3.1 Basic method

Most users (programmers) do not think of their programs as existing in one continuous linear address space. Rather they tend to think of their memory in multiple segments, each dedicated to a particular use, such as code, data, the stack, the heap, etc.

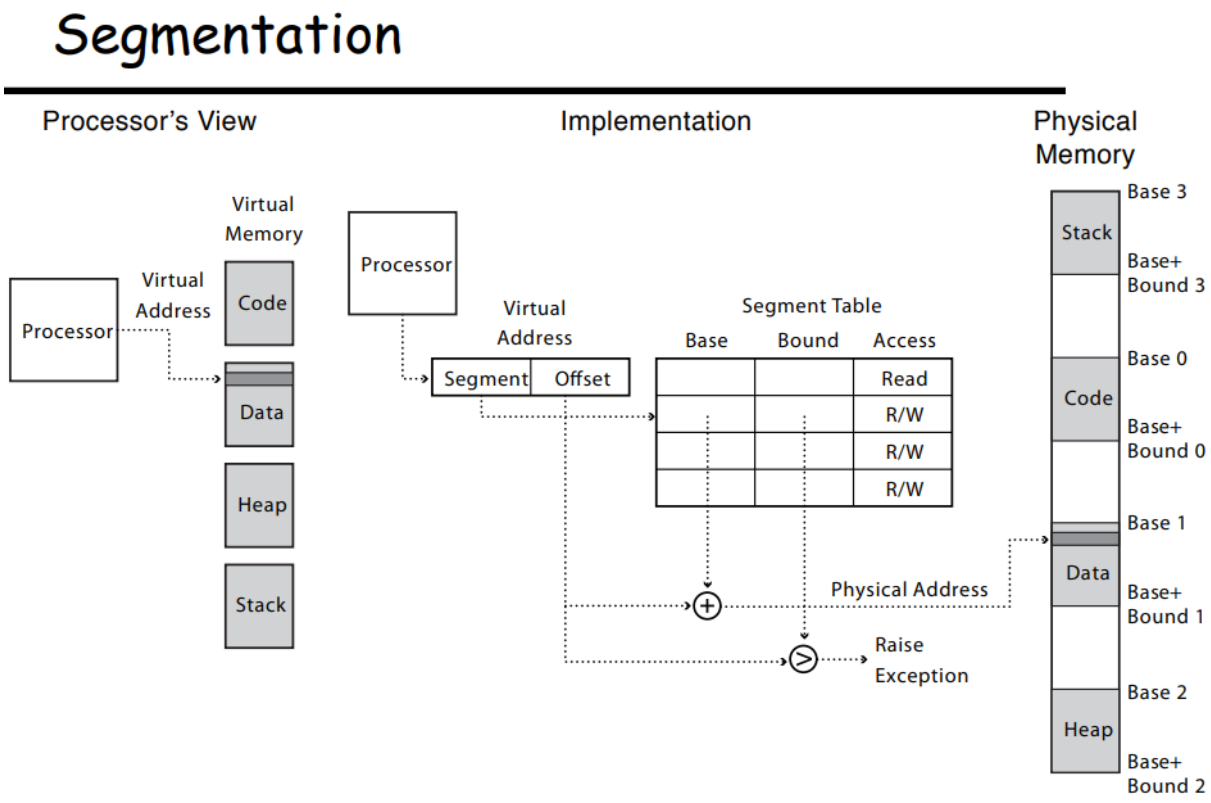
A segment can be located anywhere in the physical memory, and each have a start, length, access permission.

Memory segmentation supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment. For example, a C compiler might generate 5 segments for the user code, library code, global (static) variables, the stack, and the heap

Processes can share segments. Same start/end and same/different access permissions

8.3.2 Segmentation hardware

A segment table maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers discussed previously.



The processor sees the usual virtual address, and is sent through the memory bus to the segment table. The segment number directly access an entry in the table. The base is fetched and added to the offset, corresponding to the start of the physical address space. If we go beyond the bound we raise an exception.

How does a process context switch affect the segmentation address translation

We need to store the entire segment table for each process.

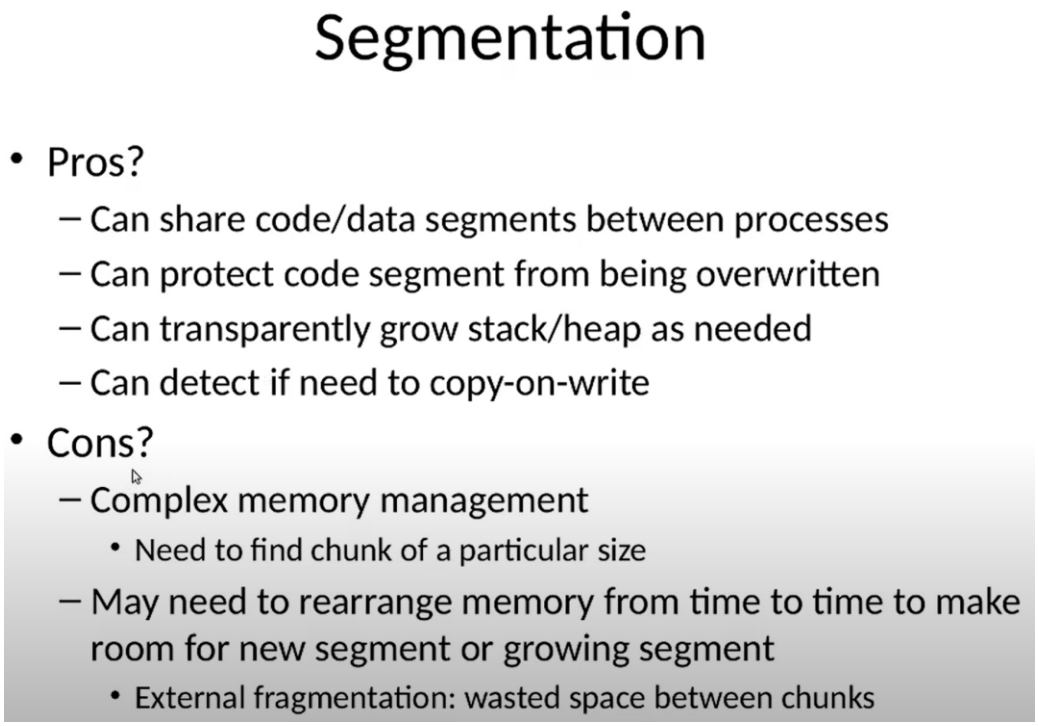
Segments also allow for a more efficient implementation of the UNIX fork. The UNIX fork makes a complete copy of the process. When using segment address translation:

- Copy the segment table into the child
- Mark parent and child segments as read only
- Start child process, return to parent
- If child or parent writes to a segment (ex stack or heap), only then we need to copy
  - Trap into kernel
  - Make a copy of the segment and continue

Zero on reference

If it is not clear, when the program begins, how much memory it will use, the OS can address this by using zero-on-reference. With zero-on-reference, the OS can allocate more memory for the stack or the heap if the program uses memory beyond the end of the stack/heap. The amount of memory can be predefined of how much we need, or a fixed size. It is also important to zero out the memory, because we don't want to use reuse some memory, map it into our process and read some old/sensitive memory. Then modify the segment table, so update position of updated memory segment.

With allocating contiguous blocks of memory, result in external fragmentation. External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the sum total could.



8.4 Paged translation

Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in equal sized blocks known as pages.

The basic idea behind paging is to divide physical memory into a number of equal sized blocks called frames, and to divide a programs logical memory space into blocks of the same size called pages. Any page can be placed into any available frame.

A page table is used to look up what frame a particular page is stored in at given moment. Each process has its own page table stored in physical memory.

Enough to save pointer to page table

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)
- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.
- Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical. The former determines the address range of the logical address space, and the latter relates to the physical address space.

Paging however suffers from internal fragmentation. Memory is allocated in chunks the size of a page. Larger page sizes waste more memory, but are more efficient in terms of overhead. Smaller pages would result in a larger page table and reduce internal fragmentation.

2.2.1 Paging – Copy on Write

We can share memory between processes by setting the entries in both page tables to point to the same page frames. In order to have a reference count and keep track of this shared memory we need to implement a *core map*. The core map track which processer are pointing to which frame.

Unix fork with copy on write

- Copy the table of parent into the child process
- Mark all pages (in new and old page tables) to read only
- Trap into kernel on write (in child or parent)
- Copy page
- Mark both as writeable
- Resume execution

2.2.2 Fill on demand

We can start running a program before all of its code is in physical memory. We do this by:

- Setting all page table entries to invalid
- When a page is referenced for the first time, we kernel trap
- Kernel brings page in from disk
- Resume execution
- Remaining pages can be transferred in the background while program is running

2.2.3 Sparse address space

- Might want many separate dynamic segments
  - Per-processor heaps
  - Per-thread stacks
  - Memory-mapped files
  - Dynamically linked libraries
- If the virtual address space is to large, we result in way to large overhead
  - 32 bits, 4KB pages → 500K page table entries
  - 64-bits → 4 quadrillion page table entries

2.2.4 Multi-level Translation

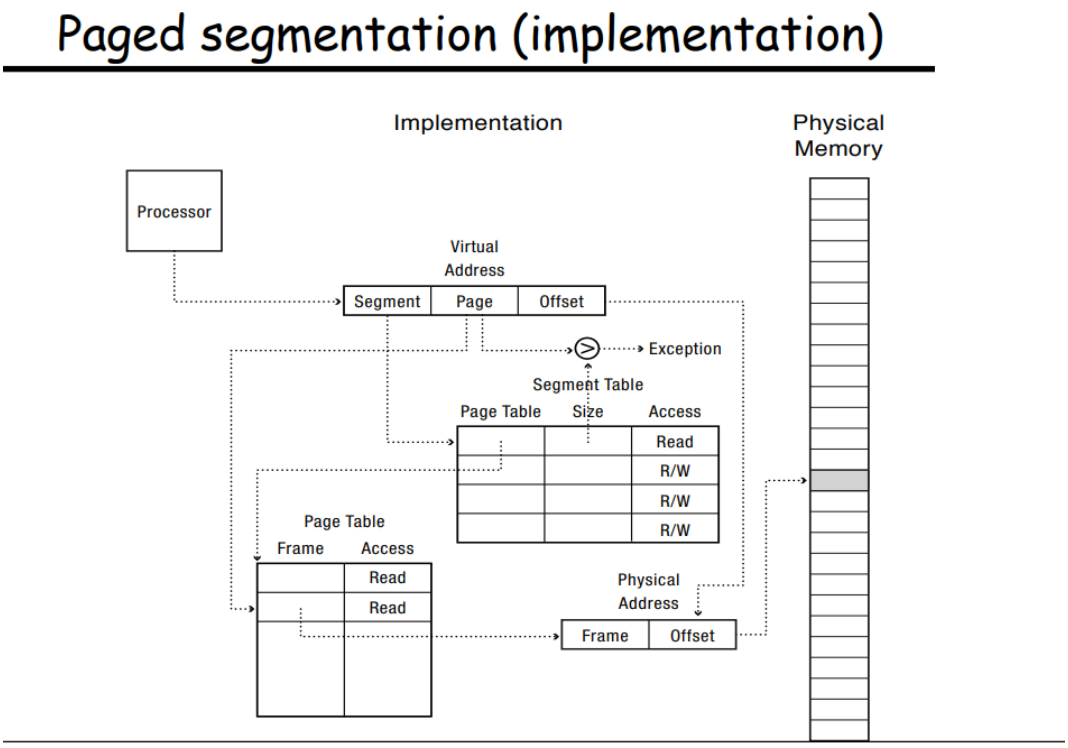
To address previous constraint we can implement multi-level translation

- Tree of translation tables (Hierarchical)
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation
- Fixed-size page as lowest level unit of allocation
  - Efficient memory allocation (compared to segments)
  - Efficient for sparse addresses (compared to paging)
  - Efficient disk transfers (fixed size units)

- Easier to build translation lookalike buffers TLB
- Efficient reverse lookup (from physical to virtual)
- Variable granularity for protection / sharing

2.2.5 Paged segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share/protection at either page or segment level



When we do a process context switch. Can simply store segment table per process, is usually done in hardware.

8.4 Portability

- Many operating systems keep their own memory translation data structures
  - List of memory objects (segments)
  - Virtual page -> physical page frame
  - Physical page frame -> set of virtual pages (for reference counting, reverse mapping)
- One approach: Inverted page table
  - Hash from virtual page -> physical page
  - Space proportional to % of physical pages

Efficient address translation

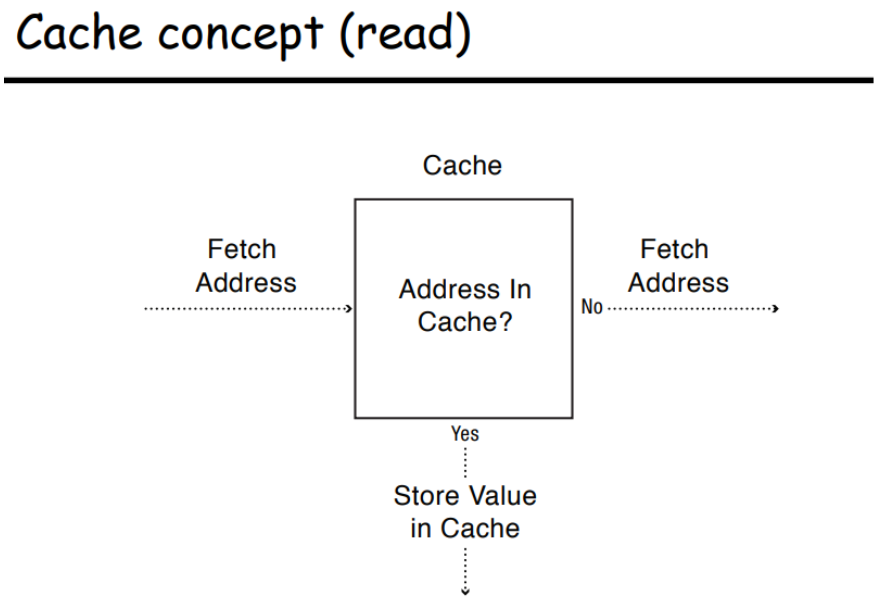
9. Cache and Virtual memory

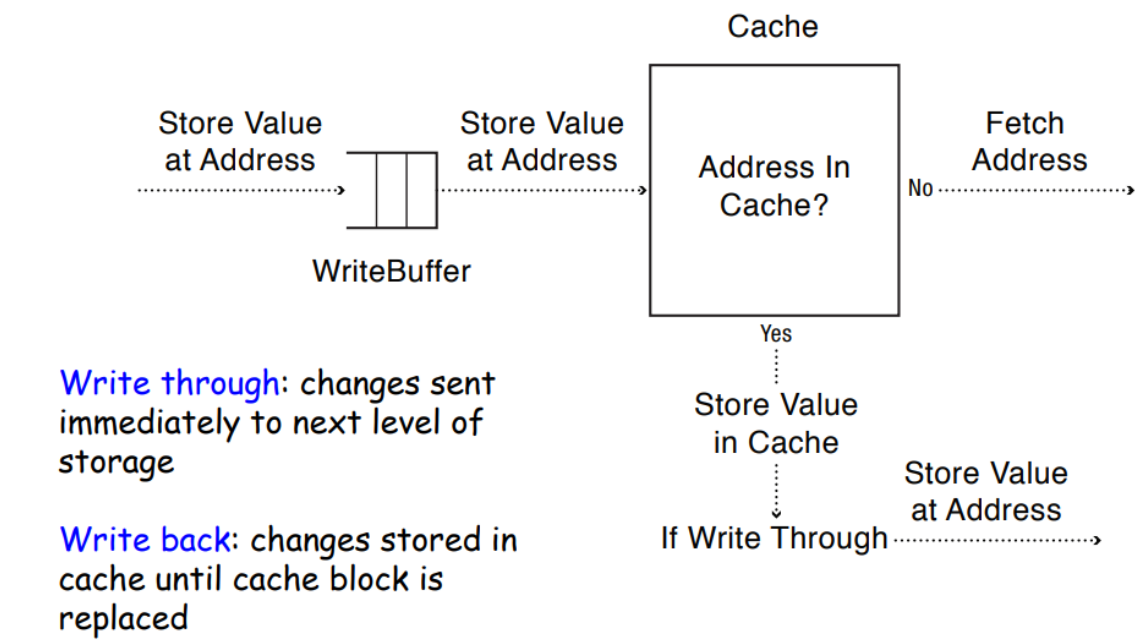
9.1 Cache concept

A *cache* is a hardware or software component that stores data so that future requests for that data can be served faster. The data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere.

We say that a cache *hits* if the cache has a copy of requested data or *misses* otherwise.

- **Cache block:** Unit of cache storage (multiple memory locations)
- **Temporal locality:** A program tends to reference the same memory locations multiple times (instructions in a loop)
- **Spatial locality:** Program tends to reference nearby locations (data in a loop)



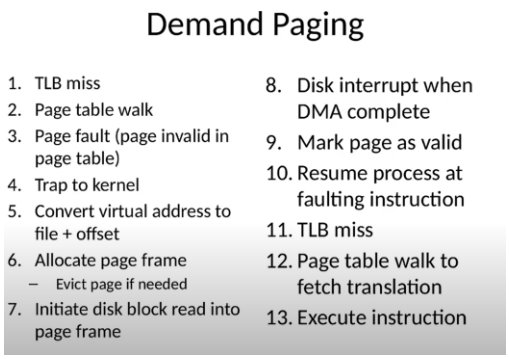


9.2 Demand paging

Preceding sections talked about how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages non-contiguously in memory. However the entire process still had to be stored in memory somewhere. In practice, most real processes do not need all their pages, or at least not all at once, for several reasons: Error handling code is not needed unless that specific error occurs, some of which are quite rare. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice. Certain features of certain programs are rarely used, such as the routine to balance the federal budget. :-)

The ability to load only the portions of processes that were actually needed (and only when they were needed) has several benefits: Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer. Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput. Less I/O is needed for swapping processes in and out of RAM, speeding things up.

There are two major requirements to implement a successful demand paging system. We must develop a frame-allocation algorithm and a page-replacement algorithm. The former centers around how many frames are allocated to each process (and to other needs), and the latter deals with how to select a page for replacement when there are no free frames available. The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance.



9.2.1 Demand paging concept

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.)
- If the process only ever accesses pages that are loaded in memory (memory resident pages), then the process runs exactly as if all the pages were loaded in to memory
- On the other hand, if a page is needed that was not originally loaded up, then **a page fault trap**is generated, which must be handled in a series of steps:
  - The memory address requested is first checked, to make sure it was a valid memory request.
  - If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
  - A free frame is located, possibly from a free-frame list.
  - A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)

- When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
- The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)

Allocating a page frame

- Select page to evict (if page table is full)
- Find all page table entries that refer to the old page (if page frame was shared)
- Set each page table entry to invalid
- Remove any TLB entries
- Write changes on page back to disk, if necessary (a write back)
- Most machines keep dirty bits in the page entry table (a method of bookkeeping modified pages)

9.3 Page modification

1.1.1 Emulating a Modified bit

- Some processor architectures do not keep a modified bit per page
  - Extra bookkeeping and complexity
- Kernel can emulate a modified bit
  - Set all clean pages as read-only
  - On first write to page, trap into kernel
  - Kernel sets modified bit, marks page as read-write
  - Resume execution

- Kernel needs to keep track of both
  - Current page table permission (read)
  - True page table permission (hardware page table, writeable, clean)

1.1.2 Emulating a recently used bit

- Some processor architectures do not keep a recently used bit per page
  - Extra bookkeeping and complexity
- Kernel can emulate a recently used bit
  - Set all recently unused pages as invalid
  - On first write/read, trap into kernel
  - Kernel sets recently used bit in internal data structure
  - Marks the page as read / write
  - Resume execution
- Kernel needs to keep track of both
  - Current page table permission (read)

  True page table permission (hardware page table, writeable, clean)

9.4 Page/Cache replacement

The basic goal of page replacement is to reduce cache misses.

In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process. However memory is also needed for other purposes (such as I/O buffering), and what happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider but we will check out *page replacement*

Page replacement: Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. There are many different algorithms for page replacement.

9.4.1 Basic page replacement

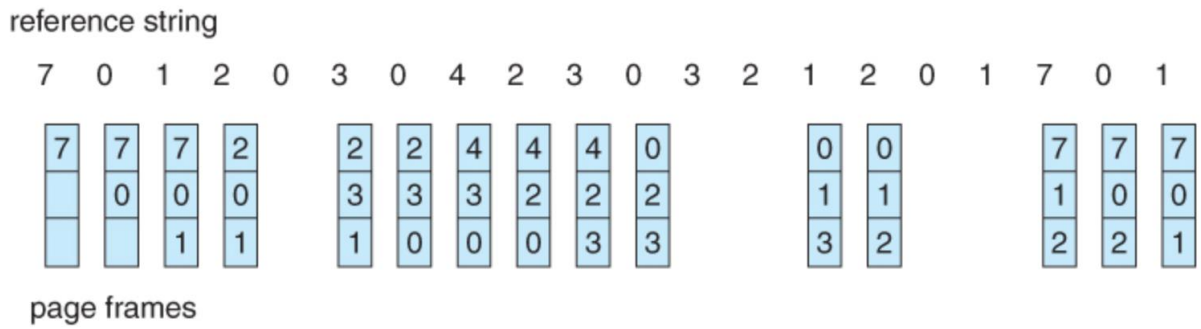
Page-fault handling must also be able to free up a frame if no free frames are available on the frame list.

- Find the location of the desired page on the disk, either in swap space or in the file system.
- Find a free frame: If there is a free frame, use it. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the victim frame.
- Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
- Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
- Restart the process that was waiting for this page.

Note that step 3 adds an extra disk write to the page-fault handling, effectively doubling the time required to process a page fault. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Otherwise the page write is required. Replacement strategies may also look for pages that does not have a dirty bit set.

9.4.2 FIFO Page Replacement

As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. Worst case scenario for FIFO is if program strides through memory that is larger than the cache. Say the cache can store 4 different tasks, if the program contains 5 different tasks, there will be a large number of evictions. A process that is spending more time paging than executing is said to be thrashing.

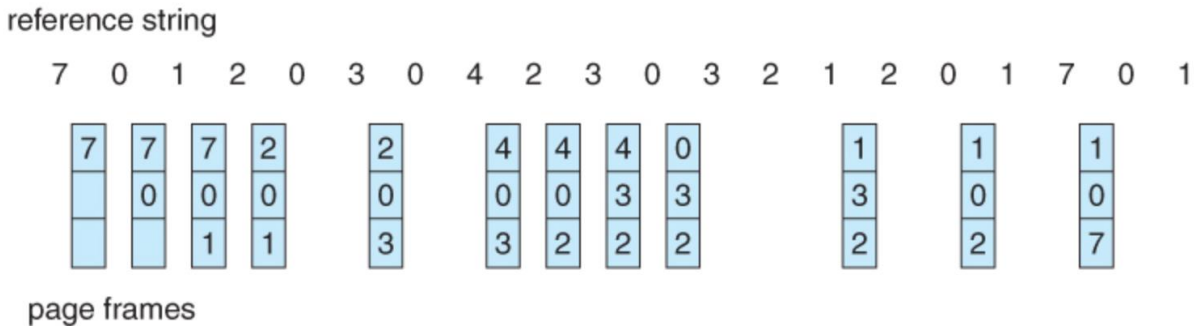


9.4.3 Optimal Page Replacement (M|IN / OPT)

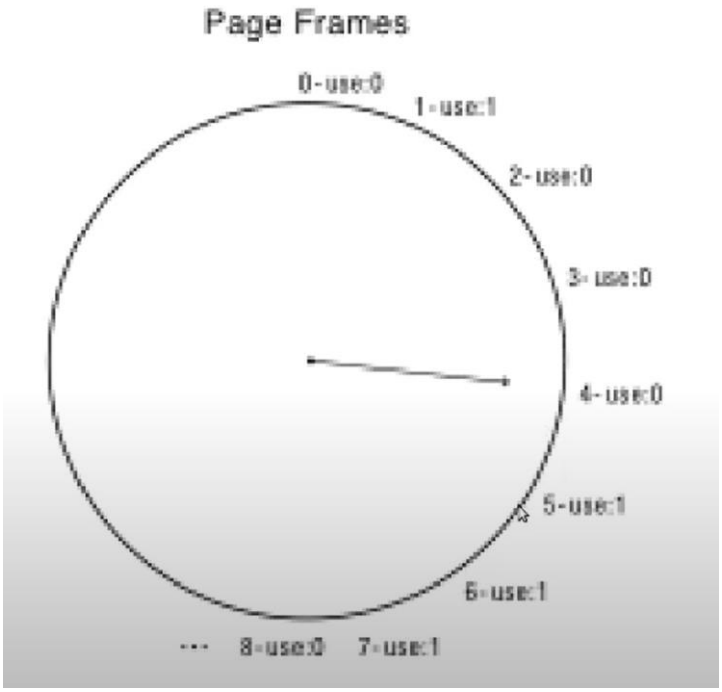
A purely theoretical algorithm. This algorithm is simply "Replace the page that will not be used for the longest time in the future." This however requires foretelling in the future, but it makes a good benchmark / comparison for other algorithms.

9.4.4 Least-Recently Used (LRU) Replacement

The prediction behind LRU, the Least Recently Used, algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest load time, and the latter looks at the oldest use time.).



Unfortunately full implementation of LRU requires hardware support, and few systems provide the full hardware support necessary. However, we can make an approximation. We begin with periodically sweeping through all pages. If a page is unused, reclaim it, else mark as unused. In particular, many systems provide a reference bit for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit of precision is enough to distinguish pages that have been accessed since the last clear from those that have not, but does not provide any finer grain of detail.

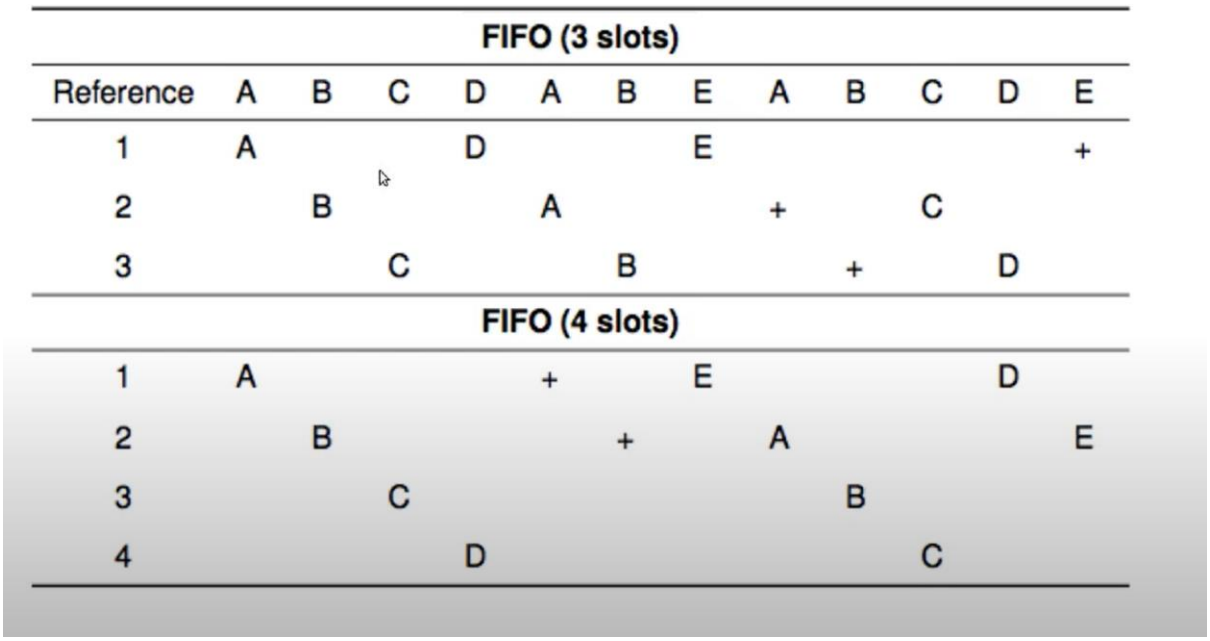


9.4.5 Least-Frequently Used (LFU) Replacement

Replaces the cache entry used the least often (in the recent past)

9.4.6 Belady's Anomaly

increasing the number of frames (caches) available can actually increase the number of page faults that occur! (Not best example under)



FIFO (3 slots)												
Reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+		D	

FIFO (4 slots)												
1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

9.5 Memory-Mapped files

Rather than accessing data files directly via the file system with every file access, data files can be paged into memory the same as process files, resulting in much faster accesses (except of course when page-faults occur.) This is known as memory-mapping a file.x

Basically a file is mapped to an address range within a process's virtual address space, and then paged in as needed using the ordinary demand paging system. Note that file writes are made to the memory page frames, and are not immediately written out to disk. (This is the purpose of the "flush( )" system call, which may also be needed for stdout in some cases. This is also why it is important to "close( )" a file when one is done writing to it - So that the data can be safely flushed out to disk and so that the memory frames can be freed up for other purposes. Some systems provide special system calls to memory map files and use direct disk access otherwise. Other systems map the file to process address space if the special system calls are used and map the file to kernel address space otherwise, but do memory mapping in either case. File sharing is made possible by mapping the same file to the address space of more than one process.

Copy-on-write is supported. The program can load/store instructions on segment memory , implicitly operating on the file. Likewise we get a page fault trap if portion of file is not already in memory.

9.5.1 Advantages of memory-mapped files

- **Programming simplicity, esp for larger files**. We can operate directly on file, instead of the regular copy in / copy out. Usually when opening a file we use explicit read / write system calls. Data is copied to user process using system call, before we can operate on it, finishing with copying the data back to kernel using system call.
- **Zero-copy I/O.** Data brought from disk directly into page frame
- **Pipelining.** Process can start working before all the pages are populated
- **Interprocess communication.** Shared memory segment vs temporary file

9.5.2 From Memory-Mapped files to Demand-Paged Virtual Memory

- Every process segment backed by a file on disk
  - Code segment -> code portion of executable (is laying on the disk)
  - Data, heap, stack segments -> temp files
  - Shared libraries -> code file and temp data files on disk
  - Memory-mapped files -> memory-mapped files
  - When process ends we can delete temp files
- Unified memory management across file buffer and process memory

10. Storage systems

Main concepts

- Storage that usually survives across machine crashes
- Block level (random) access
- Large capacity at a lower cost
- Relatively slow performance

10.1 Magnetic disks

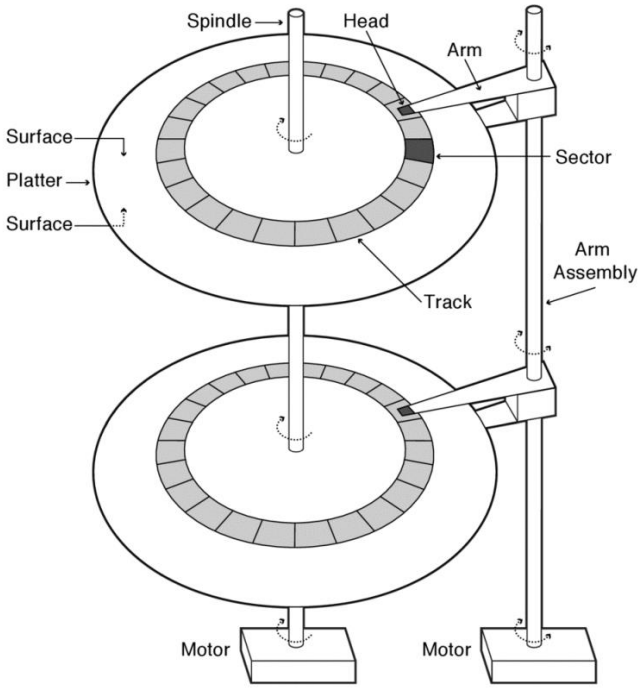
Magnetic disk is a non-volatile storage technology that is widely used in laptops, desktops, and servers. Disk drives work by magnetically storing data on a thin metallic film bonded to a glass, ceramic, or aluminum disk that rotates rapidly

The disc head is what writes and reads data from sensing or introducing a magnetic field on a surface. A head crash may appear if the head contacts the surface.

Data bits are stored in sectors, typically 512 bytes. The disk hardware cannot read or write individual bytes or words; instead, it must always read or write at least an entire sector. A circle of sectors on a surface is called a track. The disk can read/write on a track without needing to move the disk arm → much greater performance.

Disk drives often include a few MB of buffer memory, memory that the disk's controller uses to buffer data being read from or written to the disk, for track buffering, and for write acceleration.

A disk's sectors are identified with logical block addresses (LBAs) that specify the surface, track, and sector to be accessed.



10.1.1 Disk scheduling

FIFO
The simplest thing to do is to process requests in first-in-first-out (FIFO) order. Unfortunately, a FIFO scheduler can yield poor performance. For example, a sequence of requests that alternate between the outer and inner tracks of a disk will result in many long seeks.

SPTF/SSTF
An initially appealing option is to use a greedy scheduler that, given the current position of the disk head and platter, always services the pending request that can be handled in the minimum amount of time. This approach is called shortest positioning time first (SPTF) (or shortest seek time first (SSTF) if rotational positioning is not considered.) SPTF and SSTF have two significant limitations. First, because moving the disk arm and waiting for some rotation time affects the cost of serving subsequent requests, these greedy approaches are not guaranteed to optimize disk performance. Second, these greedy approaches can cause starvation when, for example, a continuous stream of requests to inner tracks prevents requests to outer tracks from ever being serviced.

Elevator, SCAN, and CSCAN
Elevator-based algorithms like SCAN and CSCAN have good performance and also ensure fairness in that no request is forced to wait for an inordinately long time. The basic approach is similar to how an elevator works: when an elevator is going up, it keeps going up until all pending requests to go to floors above it have been satisfied; then, when an elevator is going down, it keeps going down until all pending requests to go to floors below it have been satisfied.

10.2 Flash memory

Flash storage is a type of solid state storage: it has no moving parts and stores data using electrical circuits. Because it has no moving parts, flash storage can have much better random IO performance than disks, and it can use less power and be less vulnerable to physical damage. On the other hand, flash storage remains significantly more expensive per byte of storage than disks.

Flash storage access and performance

Flash storage is accessed using three operations.

- **Erase erasure block**. Before flash memory can be written, it must be erased by setting each cell to a logical “1”. Flash memory can only be erased in large units called erasure blocks. Today, erasure blocks are often 128 KB to 512 KB. Erasure is a slow operation, usually taking several milliseconds. Erasing an erasure block is what gives flash memory its name for its resemblance to the flash of a camera.
- **Write page**. Once erased, NAND flash memory can be written on a page-by-page basis, where each page is typically 2048-4096 bytes. Writing a page typically takes tens of microseconds.
- **Read page**. NAND flash memory can be read on a page by page basis. Reading a page typically takes tens of microseconds.

10.3 Flash vs Disk

Spinning disks are often used when capacity is the primary goal. For example, spinning disk is often used for storing media files and home directories. For workloads limited by storage capacity, spinning disks can often provide much better capacity per dollar than flash storage.

Flash storage is often used when good random access performance or low power consumption is the goal. For example, flash storage is frequently used in database transaction processing servers, in smart phones, and in laptops. Flash memory can also have a significant form factor advantage with respect to physical size and weight.

11. Files and Directories

File systems designers face four major challenges

1. **Performance**: File systems need to provide good performance while coping with the limitations of the underlying storage devices. In practice, strive to ensure good spatial locality, where blocks that are accessed together are stored near one another, ideally in sequential storage blocks.
2. **Flexibility:** . One major purpose of file systems is allowing applications to share data, file systems need to be able to handle several different types of files, short lived, random-access
3. **Persistence:** File systems must maintain and update both user data and their internal data structures on persistent storage devices so that everything survives operating system crashes and power failures.
4. **Reliability:** File systems must be able to store important data for long periods of time, even if machines crash during updates or some of the system's storage hardware malfunctions.

11.1 Implementation overview

File systems must map file names and offsets to physical storage blocks in a way that allows efficient access. Although there are many different file systems, most implementations are based on four key ideas: directories, index structures, free space maps, and locality heuristics

- **Directories:** Used to map human-readable files to file numbers, often a special file containing lists of file names
- **index structure:** An index structures is needed in order to locate a file blocks. Can be any datastructure that can map a file number and offset to a storage block. Often a tree
  Free space map: A free space map is a system data structure used to track which storage blocks are free and which are in use. It is also used to allocate free blocks in order to grow a file. Want to strive after good spatial locality. A file system's free space map is often implemented as a bitmap so that it is easy to find a desired number of sequential free blocks near a desired location.
- **Locality heuristic:** A file system block allocation policy that places files in nearby disk sectors if they are likely to be read or written at the same time.

A hard link can bee seen as a mirrored copy of the original file. Since a file number can appear in multiple directories, hard links are used to map different path names to the same file number. Whereas a soft link however, map a file name directly to another name or a pointer to the original file. If you were to delete the original path of the soft link, the “soft-linked-file” would not be able to be opened because the soft link wouldn’t point at anything.

A soft link has the unique ability to cross the filesystem. For example you could have a link to a directory to one hard drive inside a directory that lies on a different hard drive. If you were to delete the original path of the soft link,

11.2 Directories

File systems use directories to store their mappings from human-readable names to internal file numbers, and they organize these directories in a way that provides hierarchical, name-to-number mappings. A root directory is predefined in the file system, in order for a recursive file-searching algorithm to work, given a base case.

Directory internal

Efficient search (hash lookups, B-trees)

11.3 Files: file systems

Once a file system has translated a file name into a file number using a directory, the file system must be able to find the blocks that belong to that file. In addition to this, we also implement some target goals for improved performance.

- Support sequential data placement to maximize sequential file access
- Provide efficient random access to any file block
- Limit overheads to be efficient for small files
- Be scalable to support large files
- Provide a place to store per-file metadata such as the file’s reference count, owner, access control list, size, and access time

11.3.1 FAT Microsoft File Allocation Table (Linked list)

The FAT file system uses an extremely simple index structure — a linked list —Today: The FAT file system is still widely used in devices like flash memory sticks and digital cameras where simplicity and interoperability are paramount.

Each file in the FAT file system, corresponds to a linked list of FAT entries (an array of 32 bits entries in a reserved area in the volume), and each FAT entry has a pointer to the next entry. The FAT also has one entry for each block in the volume.

A file's number in this system is the index of the file's first entry in the FAT. In this way, given a file's number we can also find the rest of the files FAT entries and blocks (the pointer mentioned earlier).

Locality heuristic

Different implementations of FAT may use different allocation strategies, but FAT implementations' allocation strategies are usually simple. For example, some implementations use a next fit algorithm that scans sequentially through the FAT starting from the last entry that was allocated and that returns the next free entry found. This will often result in the fragmentation of a file, spreading the file's blocks across the volume rather than achieving the desired sequential layout. Can use *defragmentation tool* in order to try and prevent this.

Conclusion of FAT system

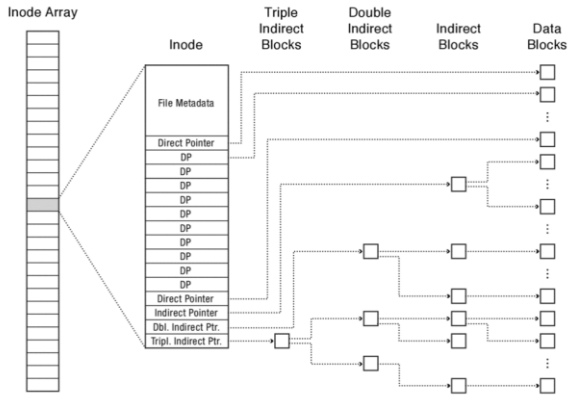
- **Poor locality:** FAT implementations typically use simple allocation strategies such as next fit. These can lead to badly fragmented files.
- **Poor random access:** Random access within a file requires *sequentially traversing* the file's FAT entries until the desired block is reached.
- **Limited metadata and access control:** Metadata does not include access control information like the file's owner, so that any user can read or write any data stored in the file system.
- **No support for hard links:** FAT represents each file as a linked list of 32-bit entries in the file allocation table. This representation does not include room for any other file metadata, each file be accessed via exactly one directory entry, ruling out multiple hard links to a file.
- **Limitations on volume and file size:** FAT table entries are 32 bits, but the top four bits are reserved. Thus, a FAT volume can have at most 2 28 blocks. With 4 KB blocks, the maximum volume size is limited (e.g., 228 blocks/volume × 212 bytes/block = 2 40 bytes/volume = 1 TB). Can support larger block size but high risk of wasting space due to internal fragmentation.

11.3.2 FFS Unix Fast File System (Fixed tree)

FFS uses a tree-based multi-level index for its index structure to improve random access efficiency, and it uses a collection of locality heuristics to get good spatial locality for a wide range of workloads.

FFS's index structure, called a multi-level index, is a carefully structured tree that allows FFS to locate any block of a file and that is efficient for both large and small files.

Index structures. To keep track of the data blocks that belong to each file, FFS uses a fixed, asymmetric tree called a multi-level index



Each file is a tree with fixed-sized data blocks (e.g., 4 KB) as its leaves. Each file's tree is rooted at an inode that contains the file's metadata. A file's inode (root) also include an array of pointers that can be used to find all of the file's blocks. Some of these pointers point directly to the tree's data leaves and some of them point to internal nodes in the tree. Typically, an inode contains 15 pointers. The first 12 pointers are direct pointers that point directly to the first 12 data blocks of a file.

The 13th pointer is an *indirect pointer* pointing to an indirect block. The indirect block is a regular block of storage but contains an array of direct pointers

The 14th pointer is a double indirect pointer which points to a double indirect block; a double indirect block is an array of indirect pointers, each of which points to an indirect block

Finally, the 15th pointer is a triple indirect pointer that points to a triple indirect block that contains an array of double indirect pointers. With 4 KB blocks and 4-byte block pointers, a triple indirect pointer can index as many as (1024)3 data blocks containing 4 KB × 10243 = 2 12 × 230 = 242 bytes (4 TB).

All of a file system's inodes are located in an inode array that is stored in a fixed location on disk. A file's file number, called an inumber in FFS, is an index into the inode array: to open a file (e.g., foo.txt), we look in the file's directory to find its inumber (e.g., 91854), and then look in the appropriate entry of the inode array (e.g., entry 91854) to find its metadata.

Important characteristics of the FFS multi level index

- **Tree Structure:** Each file is represented as a tree, which allows the file system to efficiently find any block of a file.
- **High degree:** The FFS tree uses internal nodes with many children compared to the binary trees often used for in-memory data structures. High degree nodes also improve efficiency for sequential reads and writes — once an indirect block is read, hundreds of data blocks can be read before the next indirect block is needed.
- **Fixed structure:** The FFS tree has a fixed structure. For a given configuration of FFS, the first set of d pointers always point to the first d blocks of a file; the next pointer is an indirect pointer that points to an indirect block;
- **Asymmetric:** Rather than putting each data block at the same depth, FFS stores successive groups of blocks at increasing depth so that small files are stored in a small-depth tree, the bulk of medium files are stored in a medium-depth tree, etc for large files

Free space management. FFS's free space management is simple. FFS allocates a bitmap with one bit per storage block. The ith bit in the bitmap indicates whether the ith block is free or in use. The position of FFS's bitmap is fixed when the file system is formatted, so it is easy to find the part of the bitmap that identifies free blocks near any location of interest.

Locality heuristics. FFS uses two important locality heuristics to get good performance for many workloads: block group placement and reserved space.

Block group placement: FFS places data to optimize for the common case where a file's data blocks, a file's data and metadata, and different files from the same directory are accessed together.

Reserved space: Block group placement relies on there being a significant amount of free space on the disc. FFS therefore reserves some fraction of the disk's space (e.g., 10%) and presents a slightly reduced disk size to applications. If the actual free space on the disk falls below the reserve fraction, FFS treats the disk as full

11.3.3 NTFS Microsoft New Technology File System (Flexible trees with extents)

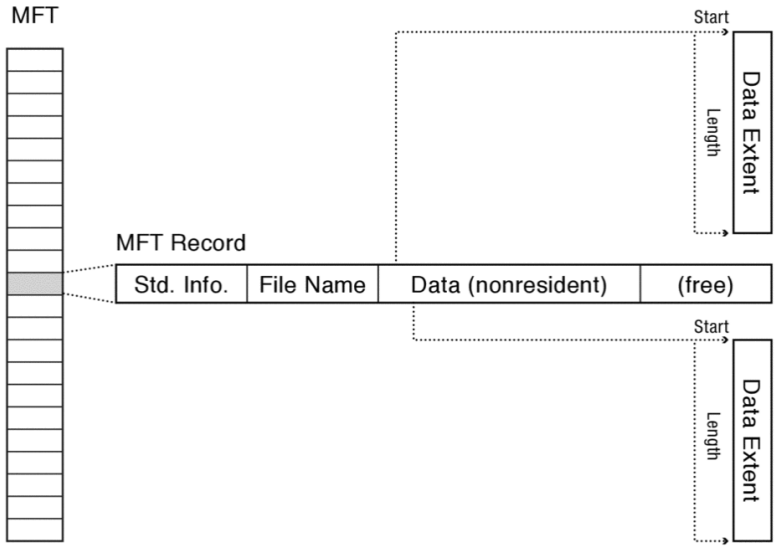
Like FFS, NTFS uses a tree-based index structure, but the tree is more flexible than FFS's fixed tree. Additionally, NTFS optimizes its index structure for sequential file layout by indexing variable-sized extents rather than individual blocks.

Index structure

Rather than tracking individual file blocks, NTFS tracks extents. An extent is a variable-sized region of a file that is stored in a contiguous region on the storage device.

Each file in the NTFS system, is represented by a variable-depth (flexible) tree. The extent pointers for a file with a small number of extents can be stored in a shallow tree, even if the file, itself, is large. Deeper trees are only needed if the file becomes badly fragmented.

The roots of these trees are stored in a master file table that is similar to the FFS's inode array. This master file table (MFT) stores an array of 1 KB MFT records, each of which stores a sequence of variable-sized attribute records. NTFS uses these attribute records to store both data and metadata, both are just considered attributes of a file.



However, some attribute records may be too large to store in a single MFT record. The MFT record therefore has to store extent pointers instead and the actual contents in those extents. This is called a non-resident attribute. A sequence of extent pointers, each of which specifies the starting block and length in blocks of an extent of data

A resident attribute just store its directly in a the master file table (MFT) record.

Locality heuristics

Most implementations of NTFS use a variation of best fit, where the system tries to place a newly allocated file in the smallest free region that is large enough to hold it. In NTFS's variation, the system caches the allocation status for a smaller region of the disk and searches that region first. If the bitmap cache holds information for areas where writes recently occurred, then writes that occur together in time will tend to be clustered together (special locality)

An important NTFS feature for optimizing its best fit placement is the SetEndOfFile() interface, which allows an application to specify the expected size of a file at creation time. In contrast, FFS allocates file blocks as they are written, without knowing how large the file will eventually grow.

NTFS extents vs FAT & FFS blocks

There are several benefits of NTFS-style extents in comparison to FAT or FFS blocks. For example does extents reduce the size of metadata needed to store when handling with blocks. In the two latter systems, each individual block would have to include some metadata itself. This affects the access time of (especially) larger files, having fairly poor performance when having to read through several layers of indirection.

NTFS extents also have a greater capacity and is stored sequentially in the memory. The size of the extents results in a file rarely have to include several extents, even for larger files. This consequentially leads to a shallower index-tree, making lookups faster and less complex.

Secondly the implementation of blocks in the FFS system, also lead to some inefficiencies when it comes to creating smaller files. A one byte file requires both an inode and a data block.

Finally, blocks are also more prone to the risk of fragmentation (specially in the FAT system). Allocation of block space is not limited to sequential blocks and can therefore be scattered around resulting in spatial locality issues (the FFS system somewhat solves this by reserving relative large space in memory for fragmentation).

11.3.4 COW Copy-On-Write File Systems

When updating an existing file, copy-on-write (COW) file systems do not overwrite data when modifying it, changes are instead written on a newly allocated block. This technology works as a version control, providing a screenshot of any given file. If you were to have a crash whilst writing, instead of overwriting both old and new memory, you would be able to retrieve the old file

12. Security

12.1 Security concepts

- The owner may wish to maintain the confidentiality of information stored in the computer system. That is, the information should not be disclosed to any person who has not been authorized to receive it.
- The owner may wish to maintain the integrity of information stored in the computer system. That is, the information should not be modified or deleted by any person who has not been authorized to do so.
- The owner may wish to preserve the availability of the services provided by the computer system. That is, persons authorized to use the system should be able to do so without interference from adversaries. The adversaries should not be able to cause a denial of service.
- The owner may wish to ensure accountability. That is, it should be possible to determine how users have chosen to exercise their authority, so that they can be held responsible for the discretionary choices they made within the limits set by the security controls.

Principal: A principal in computer security is an entity that can be authenticated by a computer system or network.

Authorization: Who is permitted to do what?

Authentication: How do we know who the user is?

Encryption Privacy and authentication across an insecure network

Auditing: Record of who changed what, for post-hoc diagnostics

12.2 Authentication

Logging into a computer system using a password at the start of a session of usage comes with several disadvanteges.

- Because the authentication takes place only at the beginning of the session. No attempt is made to notice whether you have walked away and someone else has taken your place.
- Someone can steal the password without depriving you of it
- Because the same password is used each time you log in, anyone who observes you using it can then reuse it
- Easy to remember -> easy to guess, harder to remember -> might write it down / store it

12.2.1 Principle of least privilege

In a system we often want to grant each principal the least permission possible for them to do their assigned work. In this way we minimize the amount of code that is needed to be ran inside kernel or as sysadmin. Implementing this however is difficult. It is often hard to know who needs what privilege in advance.

12.2.2 Access control matrix

An access control matrix is a security model of protection state in computer systems. Within this system, each protected resource is noted with a list of who is permitted to do what. A file or directory may implement this in the form of a list of permission: owner, group, world; read, execute, write.

12.2.3 Password capture using spoofing and phishing

Spoofing/phishing is the act disguising a communication from an unknown source as being from a known, trusted source.

One form of spoofing attack is to write a program that puts the correct image on the screen to look like a logged-out system prompting for username and password. Thus, when someone comes up to the computer and sees what looks like a login screen, they will enter their information, only to have it recorded by the program.

12.2.4 Checking passwords without storing them

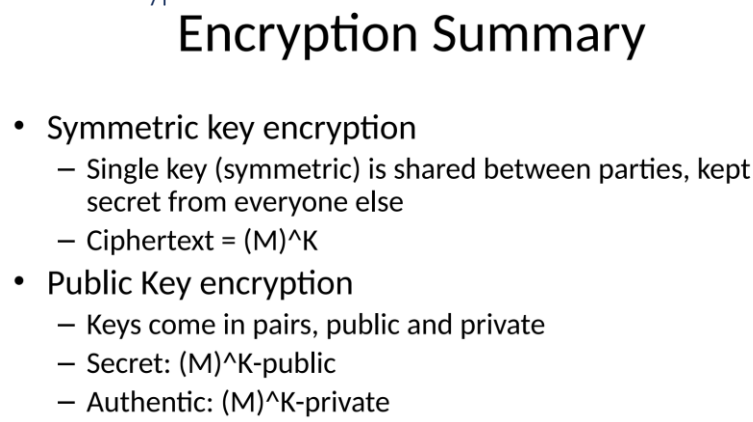
To avoid storing passwords (easily available) in a potentially vulnerable file or database, a cryptographic hash function such as the SHA-1 should be used to "cover" the passwords. Hash functions are designed not to be easily invertible. When a user tries to log into an application, the system feeds the proffered password through the same hash function and compares the resulting value with the stored hash code.

A salt may also be added to the stored password to increase the likeliness of it being cracked

A salt is a random string of characters that is added to the password before hashing and works as a safeguard for stored passwords. The salt is meant to ensure that a hashed salt is unique compared with another user who has the same hashed password. This makes it much more difficult for a

potential hacker to break the entire database by using known dictionary words or "rainbow table" to crack passwords.

12.2.5 Encryption



Public key: cryptography is the modern cryptographic method of communicating securely without having a previously agreed upon secret key, such as a password.

You begin with generating a key pair, consisting of a public key (which everybody is allowed to know) and a private key (which you keep secret and do not give to anybody). The private key is able to generate signatures which are unique, cannot be forged by anyone else without that key. Anyone who holds your public key can verify that a particular signature is genuine.

Symmetric key: Cryptography using a symmetric key, means that both entities communicating with each other, must exchange the key so that it can be used in both encryption (for the sender) and decryption (for the receiver). This encryption method differs from asymmetric encryption where a pair of keys, one public and one private, is used to encrypt and decrypt messages.

Data is firstly converted to a form which is not understandable for anyone who does not holds the secret key.

Message Digests: A message digest is a cryptographic hash function containing a string of digits created by a one-way hashing formula.

MD5: 128 bits

Secure Hashing Algorithm 1: Produces a 160 bit message digest, somewhat slower than md5 but more secure

Security Practice

- In practice, systems are not that secure
  - hackers can go after weakest link
    - any system with bugs is vulnerable
  - vulnerability often not anticipated
    - usually not a brute force attack against encryption system
  - often can't tell if system is compromised
    - hackers can hide their tracks
  - can be hard to resecure systems after a breakin
    - hackers can leave unknown backdoors

Tenex Password Attack

The password checking procedure would access certain memory pages during checking. By looking at which pages were accessed during checking of the password, user passwords could be guessed one character at a time.

Passwords were stored in clear text in directories. *Connect* System call gets access to directories. By compromising the system, you would have access to ALL passwords. Can also guess passwords in linear time

-Place first character of password as last byte on VM page
-Make sure the next page is unmapped
-Try all first letters, one by one
-Got a page fault? Must have correct first letter
-Repeat for second character of password, etc

Link about security (mainly UNIX based): http://www.scs.stanford.edu/nyu/02sp/notes/l2.pdf

12.2.6 Two-factor authentication

Rather than relying on something the authorized user knows (a password), an authentication mechanism can rely on something the user physically possesses, such as a card or small plug-in device. These physical devices are generically called tokens. The big problem with tokens is that they can be lost or stolen. Therefore, they are normally combined with passwords to provide two-factor authentication, that is, authentication that combines two different sources of information.

Another way to achieve 2FA, is by using biometric authentication; the recognition of some physical attribute of the user such as fingerprint or retina. One important difference between biometric authentication and other techniques is that it is inextricably tied with actual human identity. It will be possible to ascertain the true identity of the user. A biometric authentication method is also

prone against attacks. An adversary may record the digital coded version that is sent from the system using ex. Fingerprint.

12.3 Authorization

12.3.1 Authorization with Intermediaries

In a system we may have a set of software trusted to enforce security policy, a trusted computing base. For example is a server often needed to be trusted. A storage server can store/retrieve data, regardless of which user asks. Comes with the implication of a security flaw in the server.

12.4 Viruses and worms

Both worms and viruses strive to replicate themselves. The difference is in how they do this. A virus acts by modifying some existing program, which the adversary hopes will be copied to other systems and executed on them. A worm, on the other hand, does not modify existing programs. Instead, it directly contacts a target system and exploits some security vulnerability in order to transfer a copy of itself and start the copy running

ASLR: The memory address changes every time you run a program because of the address space layout randomization. The ASLR works as an memory protection process for the operating system. By randomizing the location where system executables are stored, malicious code can not reliably jump to exploitable snippets of code. For 32-bit systems who run 16 bits of address randomization can be defeated by a brute force attack within minutes.

Return oriented programming: Return-oriented programming (also called “chunk-borrowing à la Krahmer”) is a computer security exploit technique in which the attacker uses control of the call stack to indirectly execute cherry-picked machine instructions or groups of machine instructions immediately prior to the return instruction in subroutines within the existing program code, in a way similar to the execution of a threaded code interpreter.

A "return-to-libc" attack is a computer security attack usually starting with a buffer overflow in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process’ executable memory, bypassing the no-execute bit feature (if present) and ridding the attacker of the need to inject their own code.

12.4.1 The Morris worm

The Morris worm was a self-replicating computer program (worm) written by Robert Tappan Morris, a student at Cornell University, and released from MIT on November 2, 1988. According to Morris, the purpose of the worm was to gauge the size of the precursor “Internet” of the time - ARPANET - although it unintentionally caused denial-of-service (DoS) for around 10% of the 60,000 machines connected to ARPANET in 1988. The worm spread by exploiting vulnerabilities in UNIX send mail, finger, and rsh/rexec as well as by guessing weak passwords

12.4.2 Ping of Death

Ping of Death is a type of Denial of Service (DoS) attack in which an attacker attempts to crash, destabilize, or freeze the targeted computer or service by sending malformed or oversized packets using a simple ping command.

The concept of this attack is based on sending a ping packet larger than what was the “normal” or correctly formed IPV4 packet. When the target computer who couldn’t handle larger packet sizes would reassemble the fragments, memory overflow could occur and lead to a system crash or potentially allowing execution of arbitrary code.