CS2106 – Introduction to Operating Systems AY24/25, Y2S2 Notes

Sim Ray En Ryan

02 May 2025

- Tutorial: 2%
- Quizzes: 3%
- Lab Assignments: 25%
- Midterms: 20%
- Finals: 50%

- Prof Colin Tan (colintan@nus.edu.sg)
- Prof Anandha

Contents

1 Introduction to OS

Operating Systems are programs that act as an intermediary between a computer user and the hardware. Examples include:

- Windows
- Linux
  - Mac OS X
  - Ubuntu
  - iOS
  - Android
- Real Time OS
- Raspberry

Note that on single pipelines, when a program runs, the OS cannot run. When an OS runs, the program cannot run. Almost everything is a version of Linux.

1.1 Features and Improvements

OS adds additional overhead to any program, but in turn makes programs portable and efficient.

It must be flexible, robust, maintainable, and provide system-call interfaces. Usually written in C or C++.

For example, the OS runs Zoom, and you do not have to compile a version of Zoom for every single piece of hardware. As with old computers, you do not need to rewrite pieces of hardware to make new computations.

Multiprogramming also allows different programs (even more than the amount of CPUs) based on those that are actually using CPU, and those that are on I/O. Note that there are no system calls in kernel mode, and normal libraries (built on top of kernel mode) cannot be used. There is also no normal I/O.

1.1.1 Abstraction

Different pieces of hardware have well-defined and common functionality. Users will interact through the OS, and not be concerned with low-level, e.g, drivers, specific cards, etc.

1.1.2 Resource Allocation

OS manages the following resources:

- CPU
- Memory
- I/O

And allows for multiple programs to be executed simultaneously. It will also arbitrate conflicting requests by these multiple programs and enforce usage policies. This simplifies programming, ensures security and protection, and increases efficiency

1.1.3 Control

OS prevents errors and improper use of computers (such as access to raw sockets). It provides security and protection against accidents (badly written code) or malicious programs.

1.2 Structures and Models

1.2.1 Monolithic

The OS is one big, special program, such as in Unix or Windows. It provides:

- System Call Interface
- Process Management
- File System
- Device Drivers

The advantages include not having to switch between user and kernel mode all the time, but disadvantageous since device drivers can be buggy (and then crash the whole system). It has good performance, but is highly coupled and has the most complicated internal strucure.

1.2.2 Microkernel

The OS is small and clean. It provides:

- Inter-Process Communication
- Scheduling
- Interrupt Handlers

while the user handles:

- Process Management
- Memory Management
- File System
- Device Drivers

Whatever is done in user mode, is done in user mode. Any bugs will only affect that portion of the code. Mac OS is a microkernel. It is more robust and modular than Monolithic systems, providing better isolation between kernel and user services, but has lower performance.

1.2.3 Layered

The OS is split into multiple layers. Each layer only interacts with the layer directly above and below it. An example is:

- Application Layer
- User Interface Layer
- File System Layer
- Device Management Layer
- Kernel Layer
- Hardware Layer

1.2.4 Client-Server

A client process will request service from a service process, which is built on a microkernel. The client and server can be on different machines.

1.2.5 Exokernel

Allows access to low-level resources, and only enforces access control. The smallest kernel with lowest overhead and highest performance.

1.2.6 Models

The Windows Model is Single-User, while the Unix Model is Time-Sharing. By extension, Windows may support more GUI and usability (for a single user), while Unix may support more functionality and efficiency.

1.3 Virtual Machines and Hypervisors

Since the OS assumes total control of hardware, it may be hard to debug and monitor. A hypervisor can run various VMs such that OSs can now be monitored, and protects the rest of the hardware. A virtual machine is a software emulation of the hardware, and it believes it is the actual hardware. In MACOS, there is a program called parallels.

1.3.1 Type 1 Hypervisor

Provides individual VMs to guest OSes. Even though VirtualBox and VMWare looks like they are running on Windows, they are Type 1 Hypervisors, though they can be configured for Type 2 as well.

1.3.2 Type 2 Hypervisor

Runs on a host OS, and guest OS runs inside the VM. It is inherently slower. Intel chips allow for nested virtualization.

2 Process Abstraction

Processes have a memory context, a hardware context, and an OS context.

To switch from process A to B, the information of execution A needs to be stored, then changed to the information of B. This comes in the form of a process, task, or job, and includes:

- Code
- Data
- Registers
- Program Counter (PC)
- Other properties and resources

This is because you do not know what the other process does, and it might destroy your own execution once control is rest

2.1 Function Calls

When a function A calls another function B, it will store the current PC so that flow will resume after B finishes execution. A stack frame is used to pass parameters to the function, obtain the return value, as well as allocate any space needed for local variables.

2.2 The Stack Frame

The following is an example of a Stack Frame. The ordering is dependent on language and architecture. The stack grows when there are more functions, and shrinks when functions end. There are varying literatures as to whether it grows upwards or downwards, again dependent of architecture.

2.2.1 Stack Pointer

This points to the top of the stack region, or the first unused location. Any new stacks will be added or removed from here. Note that the values are not actually cleared, which is why you can never assume a variable contains 0.

2.2.2 Local Variables

The stack frame will allocate enough space for any local variables the function will use.

2.2.3 Parameters

The stack frame will also contain the parameters the function is given.

2.2.4 Return PC

This allows control flow to be restored after execution.

2.2.5 Frame Pointer

It points to a fixed location in the Stack Frame, and other items are accessed based on their displacement from the FP. It is not required, but extremely useful to make programs simpler.

2.2.6 Saved Registers

There are very few General Purpose Registers. The values of GPRs can be stored first in a separate location since the next function will need to use them to. This is known as register spilling, and will be restored at the end of the next function.

2.3 Example Setup and Teardown of a Function Call

2.3.1 A calls B

- Pass parameters with register and/or stack
- Save Return PC on stack

2.3.2 Execution of B

- B saves the old SP, FP, and GPRs
- B allocates spaces for local variables
- B moves the SP
- B executes
- B places the return result (if any) on stack
- B restores SP, FP, and GPRs it has used to A's state

2.3.3 Resumption of Execution of A

- A uses return result of B (if any)

2.4 Methods

2.4.1 init

The root of all processes in UNIX with a PID of 1. Is created in kernel at boot time.

2.4.2 fork

Creates a child process with an exact copy of the parents executable image and state. However, data is not shared with parent process, and they act completely independently. This function returns a 0 in the child, and returns the PID of the child in the parent. Since copying is extremely expensive, another way of sharing code is preferred unless a write is involved.

2.4.3 exec

Replaces current image with another. This only replaces the code, other information like PID remains. A child still remains a child, and a parent will still remain a parent.

2.4.4 execl

Same as exec but has variable arguments instead of one list of string of arguments. Must have NULL as last argument.

2.4.5 exit

Terminates process and returns status to parent process. Most programs have an implicit exit. Upon termination, some resources may not be released, such as PID, status, and CPU time

2.4.6 wait

The parent waits for a child process to terminate before continuing. Gets the PID of the terminated process along with its status. It also cleans up the remainder of the child system resources.

Zombie A child that exits before parent, until the parent calls wait.

Orphan A child that's parent has exited, and the child has been reparented to init.

2.4.7 Others

getpid() and main(int argc, char* argv[])

2.5 System Calls

The OS provides abstraction for low-level functions. When low-level system calls are made, the OS will switch the program from User mode to Kernel Mode. C/C++ are low level because system calls can be invoked almost directly - Libraries are just OS function wrappers. POSIX has about 100 system call functions, while Windows API has about 1000. For printf, which is a wrapper for write:

- The user invokes printf
- The library places syscall in the designated location
- The library executes code to enter kernel mode (called traps)
- The dispatcher determines the system call handler
- The system call executes and returns to library call

2.6 Exceptions and Interrupts

Exceptions are caused by problems in the program (Synchronous, such as when 1/0 is executed), while interrupts are caused by external events (Asynchronous, such as when Ctrl-C is given). Both exception and interrupt handlers will:

- Save the Register and CPU state
- Perform the handler routine, e.g., printing ”An error has occurred”
- Restore Register and CPU state
- Return, may behave as if nothing happened

2.7 Dynamically Allocated Memory

These data cannot be placed in Data as it is allocated only in runtime, and cannot be placed in Stack as it has no definite deallocation timing. (It also obviously will not be in the Text/Code section.) These data will instead be placed in the Heap, which is below Text and Data. Heap memory can create holes when memory is allocated and deallocated. Allocation may fail with external fragmentation, when the holes are too small.

2.8 Process Status

Processes change status during context switches. Can be admitted to become ready

2.8.1 New

The process has just been created, and is still initializing

2.8.2 Ready

The process lets the OS know it is ready to run. Can be switched to a running state.

2.8.3 Running

The process is running. It can be switched to a Ready state if it voluntarily gives up resources of if preempted by the scheduler. Can also be waited on to become blocked during a syscall or an I/O operation. It can also exit to become terminated.

2.8.4 Blocked

The process has to wait for an event before it can continue. Once the event occurs, it will go to a Ready State.

2.8.5 Terminated

The program lets the OS know it has finished execution and will not need resources.

2.9 Process Identification

Unique Process IDs (PID) are given to each process. PIDs may be reused, reserved, or limited depending on OS. Each process has a Process Control Block containing the following information:

- PC
- FP (if applicable)
- SP
- Memory Region Info
- PID
- State

And all of these PCBs will be stored as one Process Table.

3 Process Scheduling

The OS has a scheduler which will handle different processes executing concurrently. This can be done through virtual or physical parallelism, and with time slicing or context switching. When there are more processes than CPUs, the scheduler will solve it with a scheduling algorithm, selecting the process and allocating CPU to it.

3.1 Processing Environment

Influences scheduling algorithms.

3.1.1 Batch Processing

Batch Processing has no user, no interaction required, and does not need to be responsive. Its turnaround time is the total time taken, throughput is number of tasks finished per unit time, and CPU utilization is percentage of time CPU is working on the task.

3.1.2 Interactive/Multiprogramming

This environment requires an active user interacting with the system, and is responsive and consistent in response time. Its response time is the time between request and response, and it needs to be predictable. A periodic scheduler has a timer interrupt (every 1ms to 10ms, by the OS, that can interrupt anything else) which invokes the scheduler. It also operates on a specific time quantum, which must be a multiple of the timer interrupt.

3.1.3 Real-Time Processing

For periodic processes that have deadlines to meet.

3.2 Types

These different types have different pros and cons, regarding

- Fairness
  - Each process/user has the same CPU time
  - No starvation
- Balance
  - All parts of the computing system should be utilized

3.2.1 Non-preemptive/Cooperative

A process is scheduled until it blocks or it gives up the CPU voluntarily.

3.2.2 Preemptive

A process is given a fixed time quota to run. It is possible to be blocked or voluntarily given up. At the end of this quota, another process will get picked based on the algorithm.

3.2.3 First Come First Serve

Based on a queue, and guarantees that there will be no starvation: every task will get its chance eventually. However, average waiting time decreases by reordering based on other metrics.

3.2.4 Shortest Job First

Based on a priority queue, and picks the job with the shortest total CPU time first. This means that the OS will need to know the total CPU time for each task in advance, and it will minimize the average process waiting time.

It predicts the total CPU time using an exponential average:

$$Predicted_{n+1} = \alpha Actual_n + (1 - \alpha) \times Predicted_n$$

Where $Actual_n$ is the most recent CPU time consumed, $Predicted_n$ is based on the past history of CPU time used, and α is the weight placed on these variables.

3.2.5 Shortest Remaining Time

A variation of SJF, but uses remaining time instead. It will provide good service for short jobs even if they arrive late.

3.3 Scheduling Algorithms

3.3.1 Round Robin

Based on FCFS and a fixed time quantum. The task will run until this quantum runs out, gives up voluntarily, or blocks. It is then moved at the end of the queue, or another queue until it is unblocked. This guarantees that it will be responded to eventually (based on number of tasks and quantum). A larger quantum results in better utilization but lower waiting times, and vice versa for a smaller one.

3.3.2 Priority Based

Based on a priority queue, and processes will now have priorities. It can be both preemptive or non-preemptive. This may result in starvation for a low-priority process, especially if it is preemptive and keeps getting hijacked by a higher priority process. This can be solved by decreasing the priority after a process has been run, or enforce a time quantum.

3.3.3 Multi Level Feedback Queue

Based on a priority, based on the following rules:

- New job `->` Highest Priority
- Fully utilized time slice `->` Priority reduced
- If blocks or finishes `->` Priority retained

3.3.4 Lottery Scheduling

The scheduler gives out a limited supply of lottery tickets, and a randomly chosen process (based on number of tickets) will be given control. Processes can distribute these to their child processes, and can even control how likely more important processes will get to be executed.

4 Threading

Creating processes is expensive, and requires duplicating memory space and context. Context switching also requires saving and restoring process information. Without IPC (next chapter), also difficult for processes to pass information as they all have independent memory space. With threads, each process can have multiple parts of the program executing concurrently (in a multi-threaded process)

4.1 Threads

Each thread needs its own thread ID, registers, and stack. Other information such as code, data, and files can be shared between threads.

4.1.1 Advantages

- Multiple threads in same process requires much less resources to manage compared to multiple processes
- Threads share most of the resources, and do not need additional mechanisms to pass information
- Multi-threaded programs are more responsive, and can take advantage of multiple CPUs

4.1.2 Disadvantages

- System call concurrency can result in race conditions, deadlocks, and descychronization
- If a single thread executes exit() or exec(), the whole process terminates

4.2 Thread Models

4.2.1 User Threads

User threads are implemented as a user library, handled by the runtime system. The kernel is not aware of threads in the process. This means that processes can be multithreaded on any OS, and is more configurable and flexible. However, this means that it cannot use multiple CPUs. When one thread blocks, the process (and thus the rest of the threads) are also blocked.

4.2.2 Kernel threads

Kernel are threads implemented by the OS, and the kernel can schedule threads as well. While the kernel can now schedule these threads and execute on multiple CPUs, the thread operations are now a system call and slower. It is also generally less flexible.

4.2.3 Hybrid Models

The OS will schedule kernel threads, and user threads will bind to kernel threads. This offers the most flexibility and efficiency, but may be the most complicated to implement.

4.3 POSIX Threads

Standard defined by IEEE, can be implemented as both user or kernel thread.

- int pthread_create(...)
  - tidCreated: Thread ID
  - threadAttribute: Behavior of thread
  - startRoutine: Function pointer to function that should be executed
  - argForStartRoutine: Arguments for startRoutine
  - Return 0 for success, !0 otherwise
- void pthread_exit(...)
  - exitValue: Value to be returned
  - pthreads terminate automatically when startRoutine is reached. No return value can be obtained this way
- int pthread_join(...)
  - threadID: TID of thread to wait for
  - status: Exit value returned by target pthread
  - Return 0 for success, !0 otherwise
- Other interesting things:
  - Yielding
  - Advanced synchronization
  - Scheduling policies
  - How binding to kernel threads works

5 Inter-Process Communication

This allows for cooperation and information sharing between processes A and B, which have their own independent memory space.

5.1 Shared Memory

Process A creates a shared memory region M, and B attaches M to its own memory space. Now, A and B can communicate through M. It is efficient and easy to use, but has problems with synchronization and implementation. After finishing communication, B can detach M from memory space, which allows A to destroy M.

- shmget(): returns a shmid
  - key: Unique identifier obtained by ftok()
  - size: Size of the shared memory
  - shmflg: Flags controlling behavior
- shmid: Shared Memory Identifer, used by shmat and shmctl to attach or remove memories
- shm: A common pointer used to refer to shared memory

5.2 Message Passing

Process A prepares and sends a message M to B. B then receives this message. This system is usually provided as system calls by the OS. The synchronization can be synchronous (wait until message is received), or asynchronous (return immediately when no message received). Message passing is portable and easier to synchronize, but inefficient and harder to use.

5.2.1 Direct Communication

Send(B, msg) and Receive(A, msg). Both A and B need to know the identity of the other party, and each pair has a link.

5.2.2 Indirect Communication

Send(MB, msg) and Recieve(MB, msg). One Mailbox can be shared between multiple proceeses

5.3 Pipes

The Unix shell provides the — symbol to link input and output channels of processes. The pipe will maintain a buffer with the following synchronization:

- Writers wait when buffer is full
- Readers wait when the buffer is empty

These pipes can have multiple readers and writers, and can also be half or full duplex. Pipes can also make use of stdin, stdout, and stderr.

5.4 Signals

Signals are an asynchronous notification regarding an event sent to a process or thread. The recipient must handle it by a default set of handlers or a user-supplied handler. Common signals include:

- Kill
- Stop
- Continue
- Memory Error
- Arithmetic Error

6 Synchronization

Execution of a single sequential process is determinisitc, but is not for concurrent processes.

6.1 Race Conditions

Execution outcome depends on order shared resources are accessed or modified, also known as race conditions. For example, two concurrent branches do x += 1000. This can be broken to:

- Load x to R1
- Add 1000 to R1
- Store R1 to x

If these branches occur in the following way (example), x will be incremented by 1000 and not 2000.

- Load x to R1
- Load x to R1'
- Add 1000 to R1
- Add 1000 to R1'
- Store R1 to x
- Store R1' to x

6.2 Critical Selections

To prevent race-conditions from creating undefined behavior, critical sections of a code can be made. A Critical Section has the following properties:

- Mutual Exclusion: At any point in time, only one process can execute the critical section.
- Progress: If there is no process in CS, then another waiting process should take over
- Bounded Wait: After P_i requests for CS, there is an upperbound on the number of times the processes will use the CS
- Independence: Processes not executing in CS will never block other processes

6.2.1 Implementation

- void EnterCS (int* lock) while TestAndSet(Lock) == 1; // Lock CS
- void ExitCS (int* Lock) *Lock = 0; // Unlock CS

This works, but other processes have to keep checking the lock condition, wasting processing power especially if the time quantum is long. This is called busy waiting.

- while (Lock != 0);
- Lock = 1;
- Do stuff
- Lock = 0

Now, Lock is a race condition and may result in both processes entering a CS at the same time.

- Disable Interrupts
- while (Lock != 0);
- Lock = 1;
- Do stuff
- Lock = 0
- Enable Interrupts

Still same, as interrupts are only between the same CPU. This is also a bad solution, since interrupts may never be re-enabled. Almost all IO depends on interrupts.

- while (Turn != 0);
- Do stuff
- Turn = 1; // And vice versa

Not independent: If the first task never finishes CS, the second task never will.

- Want[0] = 1;
- while (Want[1]);
- Do stuff
- Want[0] = 0; // And vice versa

Can result in a deadlock. Deadlock occurs sometimes, depending on OS interleaving.

6.2.2 Peterson's Algorithm

- Want[0] = 1;
- Turn = 1;
- while (Want[1] && Turn == 1);
- Do stuff
- Want[0] = 0; // And vice versa

This works, but employs busy waiting and has a very low level implementation. Not general and not easy to extend to more than two processes.

6.2.3 Semaphores

Provide a way to sleep processes, and unblock or wake up sleeping processes. Found with #include ¡semaphore.h¿

- Wait (S)
  - If S is less than 0, block
  - Decrement S
- Signal (S)
  - Increment S
  - Wakes up one sleeping process

There are general (counting) semaphores, and binary semaphores (mutex). A binary semaphore can mimic the behaviour of a general semaphore

6.3 Synchronization Problems

6.3.1 Producer Consumer

Producers produce items to insert in buffer, only when buffer is not full. Consumers remove items from buffer, only when it is not empty. With semaphores, there is no busy waiting, and will be signalled by other semaphores (producer or consumer) when it is ready.

6.3.2 Reader Writers

Writers must have exclusive access to a data structure, while a reader can always access it with other readers.

6.3.3 Deadlock

When all processes are blocked. The OS can detect this and build in mechanisms to prevent it.

6.3.4 Livelock

When processes keep changing state to avoid deadlock, but still make no progress. This is undetectable by the OS as it just sees two processes running.

6.3.5 Starvation

When a process is blocked forever.

6.4 Implementation

6.4.1 Mutex

pthread_mutex, pthread_mutex_lock, pthread_mutex_unlock

6.4.2 Conditional Variables

pthread_cond_wait, pthread_cond_signal, pthread_cond_broadcast

7 Memory Abstraction

7.1 Memory Basics

Memory can be treated as a giant array of bytes, and each array has a unique index known as its physical address. The physical memory is contiguous. Memory usually come in the following two types:

- Transient: Memory that will be removed after the process ends
- Persistent: Memory that will be stored somewhere.

7.1.1 Memory Hierarchy

Storage Type	**Speed**	**Size**	**Cost**
CPU Registers	Very Fast (1 ns)	512 Bytes	Very Expensive
Cache	Fast (10 ns)	12 MB	Very Expensive ($5/MB)
RAM	Moderate (100 ns)	8 GB	Expensive
Hard Disk	Slow (10 ms)	2 TB	Inexpensive ($0.5/MB)
Off-Line Storage (Tape, Cloud)	Very Slow (Seconds)	PBs	Least Expensive

CS2106 focuses on RAM: Random Access Memory.+++

7.1.2 Operating System's Job

The OS allocates, manages, and protects memory spaces for processes. It also provides related system calls, as well as manage its own memory use.

7.2 Memory Abstraction

Without memory abstraction (using physical addresses directly), there is no management required. However, another process:

- May refer to the wrong absolute addresses: Can be fixed using address relocation. This results in slow loading time, and it is difficult to distinguish memory references from normal integer constants (that should not be updated).
- Can use a special register known as a base register and limit register. During compilation, all memory are compiled as offset from this register. All memory access is also checked against limit to protect memory space integrity.
- To check, this means every memory access incurs an additional comparison, which is slow, but does provide abstraction and segmentation.

Logical addressing is how a process views its memory space, and each process has a self-contained, independent logical memory space.

7.3 Contiguous Memory Management

Processes must be in memory during execution. Assumptions are that each process occupies a contiguous memory region, and that the physical memory is large enough to contain one or more processes with complete memory space. This is to support multi-tasking. When the memory space is full, memory is freed by removing terminated processes, or by swapping blocked processes to a larger storage.

7.3.1 Memory Partitioning

Fixed-Size Partition is where physical memory is split into partitions, and a process will occupy one of these. An advantage is that it is easy to partition and manage, but creates a lot of internal fragmentation. The partition needs to be large enough to contain the largest process.

Variable-Size Partition is where partitions are created based on size of process. OS takes more power to keep track of this and splits and merges as necessary. While there is less fragmentation, there will still be external fragmentation. It also takes more time to find where to allocate the memory using allocation algorithms, as well as compacting.

Allocation algorithms

- First Fit: Find the first hole that is large enough. OS may use a linked-list with first fit. The linked list contains:
  - Marked as free or used
  - Start of memory region
  - Size of current memory region
  - Pointer to next struct
- Best Fit: Find the smallest hole that is large enough
- Worst Fit: Find the largest hole

Once the space has been found, the hole of size M has to be split into partitions, N and M - N.

Compaction Moving occupied partitions around to create consolidated holes. Cannot be invoked too frequently as it is time consuming, processes cannot be used during compaction, and no effective work is being done.

Buddy System Only partition in powers of two. Two halves form sibling blocks, and can be merged when both are empty.. The idea is to keep used and unused spots together. With this system, an array a[0..k] can be used instead of a linked list. a[j] itself is a linked list, pointing to the starting address for memory blocks the size of 2^j. If all memory blocks are the same size (and are very small), then this solution is not that good.

Buddy system uses best fit. If a such a free block exists, remove that free block and allocate it. Otherwise, go power of twos up until a free block is found, then split until the perfect size is found.

During deallocation, if the buddy of the free block is also free, merge them repeatedly, then insert back into the free space. Buddy blocks have the exact same address only differing in one bit. If two buddy blocks are of size 2^J, then they differ on the jth bit.

Note that there is now internal and external fragmentation, but they are much easier to manage.

8 Disjoint Memory Schemes

Now, each process no longer needs to be in a contiguous memory partition.

8.1 Paging Schemes

In the physical memory, split the region into fixed size (decided by hardware, known as physical frames). Logical memory of a process is split into regions of the same size (known as logical page). This means that logically, the memory is contiguous, but may actually be split in the physical memory.

8.1.1 Page Table

As opposed to contiguous (just knowing starting address and size), disjoint memory systems need a lookup table to provide the translation from a logical page to its physical location. Logical addresses are reconstructed into page number and the offset, and the page table provides the mechanism to translate the page number into the frame number. Since frames and pages are the same size, the offsets are the same. Frame sizes are also kept as powers of two.

Given frame sizes of 2^n, in a logical address of m bits, the page number is the first $m-n$ bits and the offset is the remaining n bits. The page number and the frame number can be translated using the page table, and the physical address is frame number $\times\, 2^n +$ offset. The last frame may have internal fragmentation (on average half full), but now there is a clear separation between logical and physical addresses, allowing for great flexibility at the cost of a translation mechanism.

Page tables will now allow several processes to share the same physical memory frames, if they are both pointing to the same location. This allows for shared code for libraries or system calls, and implement "copy on write" for parent and child processes.

Pure software implementation Store page table information in PCB. There is improved understanding as memory context of a process is now the page table. However, two memory accesses are needed for every memory reference now: To check the page table, then to check actual memory. Since the page table is accessed so frequently, it can be cached in the hardware.

Hardware support The Translation Look-Aside Buffer acts as a specialized cache for a few page table entries. TLBs are part of the hardware context of a process. When a context switch occurs, TLB entries are flushed, and there will be a lot of cache-misses when a new process takes over.

Protection Schemes

Access Right Bits Each page table has a wrx, and memory access is checked against the access rights.

Valid Bit This indicates whether a page is valid to access, and out of range accesses will be caught by OS. For example, when the page size is 16 bytes but the program is only 8 bytes: The logical address is valid but not valid in the scope of the process.

8.2 Segmentation Schemes

Since different processes's memory spaces have different memory regions (Text, Data, Heap, and Stack). The Heap and Stack can grow and shrink at execution time, while the Text and Data do not. Logical memory spaces of a process are now segmented into segments, each having their own name and limit. All memory references are now specified by their segment name (segment id) plus their offset. The segment id will lookup the segment's base address and its limit in a segment table, then check Offset < Limit for validity, and then find the physical address Base Address + offset. When the validity check fails, you get an addressing error, and then a segmentation fault.

With segmentation, each segment is independent contiguous memory space, can grow, shrink, and be protected or shared independently. However, it requires variable size contiguous memory regions, and causes external fragmentation.

8.3 Paging and Segmentation Schemes

A segment is a a part of a process, and comes in different sizes. Segmentation with paging involves paging these segments further. This means that in total, there are:

- Global Data page table
- USer Code page table
- Heap page table
- Stack page table

And lookups are now 2-step. CPU generates a triplet of Segment ID, Page Number, and offset. The segment ID in the segment table returns the page table base and its limit. Then, the page number is used in the page table to obtain its validity and the frame number. With the frame number and offset, the physical address can be calculated.

9 Virtual Memory Management

Used since the logical memory space of a process, or multiple processes can be larger than physical memory. Since the secondary storage capacity is much larger than the physical memory capacity, chunks of the process that are being used are stored in physical memory, while the rest are stored on secondary storage.

9.1 Implementation via extending paging scheme

The page table now contains a "Is Memory Resident" bit in the page table entry, and can store a page to a memory resident of a non memory resident. The CPU can only access resident pages, and returns a page fault when CPU tries to access non memory resident pages. This is when the OS brings the non memory resident page needed into physical memory.

- Check if page table is a memory resident. If it is, access that location, otherwise
- Trap to OS
- Locate the page needed in secondary storage
- Load that page to physical memory
- Update page table
- Loop

If memory access often results in a page fault, thrashing happens. Since secondary storage access time is much larger than physical memory access time, this decreases efficiency greatly. However, thrashing is unlikely to happen based on temporal and spatial locality, and since most program only spend time on a small part of code, and access data in a short time period.

Temporal Locality Memory address which is likely to be used again

Spatial Locality Memory address close to a used address is likely to be used
With pages, the page will likely be accessed again in near future, and contains locations that might be accessed next.

9.2 Demand Paging

A process starts with no memory resident page and only loads pages when there is a page fault. This allows processes to start quickly and have a small memory footprint, but also means that execution will be slower when it suddenly has many page faults, and may cause thrashing effects on other processes. This also means that each process only loads the pages that it needs.

9.3 Page Table Structure

Page tables need to be stored too, which means that huge page tables result in large overhead, and the page table itself may be fragmented.

9.3.1 Direct Paging

The current paging algorithm so far, keeping all entries in one table. With 2^P pages in logical memory space, P bits are used to specify a unique page, and there are 2^P page table entries

9.3.2 2-Level Paging

The main page table holds information to other page tables. Since a process may not use the entire virtual memory space, the page table can be split into regions, and only regions used are allocated. This means the entire page able does not need to be loaded.

If the original page table has 2^P entries, with 2^M smaller page tables, M bits are needed to specify the page table, and each small table contains $2^{(P-M)}$ entries. Then, the main page directory contains 2^M entries to locate the small page tables. Can be extended into 3, 4, or as many level paging as needed. Now, the CPU gives a page directory number to lookup the page directory, a page number to look up the associated page directory, returning the frame number, which can then be used with the offset.
The page directory incurs a size overhead which is the number of entries multiplied by the size of each entry, and a time overhead due to the hierarchical paging scheme.

9.3.3 Inverted Page Table

Keeping a single mapping of physical frames to a PID and a page number. Page numbers are not unique between processes, so PID is added. The CPU outputs a PID, Page number, and offset to get the frame number, which can be added with the offset. This means that there is only one table needed for all processes (for all frames), but the entire table needs to be searched for the correct page, resulting in slower translation.

9.4 Page Replacement Algorithms

If there is no free memory during a page fault, one has to be evicted. Pages can be either clean or dirty (modified), which indicates if a write back is needed. A good algorithm reduces the total number of page faults.

9.4.1 Memory Reference Strings

Only the page number is important in comparing replacement algorithms. The memory reference string is just a list of numbers which show which frame needs to be referenced.

Then, the number of page faults can be compared. $T_{access} = (1-p) \times T_{mem} + p \times T_{pagefault}$. Since a page fault takes much longer than a memory access, the probability p should be reduced enough to make T access reasonable.

9.4.2 Optimal Page Replacement

Replaces the page that will not be used again for the longest period of time. This guarantees the minimum number of page faults, but is not realizable as it needs future knowledge of all memory references. However, the next few algorithms are designed to be as close to OPT as possible. Note that all replacement algorithms will get an initial page fault, or a cold miss as in CS2100 cache misses.

9.4.3 FIFO Page Replacement

Replace the oldest memory page. The OS maintains a queue of resident page numbers, removes the first page in the queue if replacement is needed, and updates the queue on page faults. No hardware support is needed. Does not take into account and exploit temporal locality, and has a lot of page faults.

Belady's Anomaly For FIFO, increasing the number of frames stored may increase the amount of page faults. For example, the sequence 1 2 3 4 1 2 5 1 2 3 4 5 increases from 9 to 10 page faults when frame increases from 3 to 4.

9.4.4 Least Recently Used

Replace page that has not been used in the longest time. This exploits temporal memory since you expect a page to be reused in a short time window. LRU generally approximates OPT well, and does not have Belady's Anomaly. However, it is harder to implement since one needs to keep track of the last access time. This is usually done through time approximation algorithms:

Counter A counter is incremented on every memory reference, and the current counter is stored on every reference. During replacement, replace the page with the smallest time of use. This means that all pages needs to be searched on every memory reference, and time of use may overflow.

Stack If page X is referenced, remove any current entry with entry X, and push it on top of the stack. Remove the page at the bottom of the stack if it is not found, and push X on the top of the stack. Note that this is not a pure stack as things can be removed from any position, and is thus harder to implement.

9.4.5 Second Chance Page Replacement

Modified FIFO but use a circular queue instead. Each page table entry now has a reference bit. The oldest FIFO page is selected. If the reference bit is 1, go to 0, and go next. Else, remove the page if its reference bit is 0. Set new reference bit to 0, and only set it to 1 when it is referenced again later. When all bits are 1, it is the same as FIFO. For CS2106, after loading, move pointer to the next location.

9.5 Frame Allocation Policies

Allocating N physical memory frames to M processes. Local replacement occurs when a process replaces pages among its own pages, and global replacement occurs when another process evicts their frame. The main idea of frame allocation policies is to reduce trashing, since bringing in new pages is still the most intensive operation.

9.5.1 Local Replacement

The frames allocated to a process remains constant, and performance is stable between multiple runs. However, if the frames allocated is not enough, the process' progress can be hindered. Trashing is limited to one process, but that single process can take all the I/O and still cause trouble for other processes.

9.5.2 Global Replacement

Allows self adjusment, and processes that needs more frames can dynamically get it. However, badly behaved processes can then take all frames, and there is less stability. May cause cascading trashing

9.5.3 Equal

Each process gets N / M frames.

9.5.4 Proportional

Each process gets $\frac{\text{size of process}}{\text{size of all processes}} \times N$ frames

9.5.5 Working Set Model

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

The set of pages needed is fairly constant. In a new locality, a process causes a page fault for a set of new pages, and will have very few page faults for a while. Thus, the amount of frames to allocate should be enough within an interval of time to reduce chance of page fault. If the interval is too large, it may contain pages from a different locality. If it is too small, it may miss pages in the current locality.

For the above example, an interval of 5 may be good, and may change to 2 when it hits the 1 2 1 2 sequence.

10 File System Introduction

10.1 Why

Physical memory is volatile, and external storage is used to store persistent info. However, direct access to storage media is not portable. The file system provides:

- An abstraction of the physical media
- A high-level resource management scheme
- Protection between processes and users
- Sharing between processes and users

10.2 Requirements

- Self contained: Information stored on media is enough to describe entire organization, and is plug-and-play.
- Persistent: Beyond the lifetime of the OS and processes
- Efficient: Minimum overhead and good management of free and used space

10.3 Differences with Memory Management

	Memory Management	**File System Management**
Underlying Storage	RAM	Disk
Access Speed	Constant	Variable disk I/O time
Unit of Addressing	Physical memory address	Disk sector
Usage	Address space for process (Implicit when process runs)	Non-volatile data (Explicit access)
Organization	Paging/Segmentation (determined by HW & OS)	Many different FS: ext* (Linux), FAT* (Windows), HFS* (Mac OS), etc.

Table 1: Comparison between Memory Management and File System Management

10.4 File System Abstraction

An OS exposes file system operations using system calls.

10.4.1 File

A file is a logical unit of information. It contains metadata (file attributes: Name, Identifier, Type, Size, Protection, Time, Date, Owner, ToC), the data itself, access methods, and operations. For an open file, there is a file pointer, its disk location, and an open count (how many processes has a file opened).

10.4.2 File Name

Different FS have different rules. The most common has Name.Extension, and the extension may include the filetype.

10.4.3 File Types

Each file type has an associated set of operations, and maybe a program used for processing (e.g. MS Word for .docx). Common filetypes include regular, directories, and special (character / block oriented) files.

Regular files include ASCII (Human Readable) and binary files (which need a specific program to be processed). They can be checked either via extension, or by embedded information such as with a magic number.

File Protection RWX, Append, Delete, List (Read Metadata). Most common approach is to restrict based on user identity with ACLs (List of users). It is very customizable but it becomes big. For UNIX, it is owner, group, universe. An extended ACL is ...?

File Metadata can be renamed, and along with the previous fields, also has file creation time.

File Data is an array of bytes. It can be read using sequential access, or random access. There are two more:

Fixed Length Records are an array of records that can grow or shrink, also with O(1) time.

Variable Length Records are more flexible but harder to locate.

File Operations Create, Open, Read, Write, Reposition (Seek), Truncate. Since several processes can operate on the same file, and several files can be opened at the same time, there are two approaches:

System Wide Open File Table One entry per unique file for all processes

Per Process Open File Table One entry per file used in the process, and each entry points to the system wide table.

10.5 File System Implementation Approaches: Directories

Logically groups files together. Allows the system to keep track of files too.

- Single Level: There is one main directory. Harder to manage files, but there will only be one of each file.
- Tree Structure: There can be multiple of each file, in different sub directories, since the absolute file path is different.
- Directed Acyclic Graph: Have the ability to jump sub-directories with alias, links, or shortcuts. Useful so that only one actual file exists. Links can be hard or symbolic:
  - Hard Link: Linked to file only: A point to the actual file. If A owns F, and B wants F, A and B both have separate pointers to actual F on disk. This results in lower overhead, but deletion problems when either one decides to delete F.
  - Symbolic Link: Can be a file or directory: A point to someone who has the actual file. Can result in infinite loops. In traversal, these links are often not taken into account. If A owns F, and B wants F, then B creates a link file G which finds out where is F, and accesses F. This results in simpler deletion. Deleting F has G remain (although it doesn't work), and deleting G has F remain. However, there is now a larger overhead with the special link file.
- General Graph: Like DAG but can be cyclic. Some OS ban this since it causes problems with directory traversal such as with locate, or with library calls.

There are two ways to refer to a file: Absolute file name, or relative file name (relative to CWD).

11 File System Implementation

File systems are stored on storage media (hard disk). The general disk structure is a 1-D array of logical blocks (smallest accessible units), and each logical block is mapped to a disk sector.

A good file implementation must be able to keep track of logical blocks, allow for efficient access, and utilize disk space effectively (minimizing internal fragmentation).

11.1 Disk Organization

11.1.1 Master Boot Record

The MBR is at sector 0 with a partition table, and followed by one or more partitions that are each a file system. A file system contains:

- OS Boot Up Information

- Partition Details: Number of blocks, number and location of free blocks.
- Directory Structure
- Files Information
- Actual File Data

11.1.2 Contiguous Block Allocation

Allocate consecutive disk blocks to a file. This is simple to keep track and has fast (direct) access, but results in external fragmentation.The start and length is required.

11.1.3 Linked List

Each disk block stores the next disk block number and the actual file data. Only the start block and end block needs to be saved. This solves the fragmentation problem, but random access in a file is very slow. One incorrect pointer also destroys the whole system.

11.1.4 Linked List 2.0

Have a File Allocation Table (FAT), and move all block pointers to FAT. Lookups are faster and used by MS-DOS. Each block is a lookup table to the next location, and possibly -1 for the EOF. However, an entry for every block means the FAT can become huge.

11.1.5 Indexed Allocation

Each file has an index block. IndexBlock[N] points to the Nth block address. This results in lesser memory overhead compared to FAT since only opened files are captured, while still maintain fast access, but incurs a index block overhead as well as a maximum file size based on maximum index block size. Can point to -1 if the file size is less than the amount of entries in each index block.

Increasing the limit A linked list of index blocks (chaining) allows sizes to increase. Multilevel indexes can also be used. This means there's fast access to direct data, and not so bad access to indirect data.

11.2 Free Space Management

On allocation, remove the disk block from free space list. On free, add the free disk block to free space list. This can be done with a bitmap, and any block that is free has 1 instead of 0. This provides a good set of manipulations, but the bitmap can also become big and be stored in memory.

11.2.1 Linked List

Each disk block contains the number of free disk blocks and a pointer to the next free space. This makes it easy to locate free blocks, and only the first pointer is needed in memory. However, there is high overhead as it contains a lot of addresses.

11.3 Directory Structure

A directory needs to keep track of files in a directory. Thus, it maps file names to file information.

11.3.1 Linear List

Each entry represents a file. Stores minimally a file name, file info or pointers to file information. This requires a linear search, but can be mitigated using cache to remember the latest searches.

11.3.2 Hash Table

Hash a file name to the hash table. This results in fast lookup, but needs a good hash function and good table size.

12 File System Case Studies

12.1 Microsoft FAT

There is a Master Boot Record (MBR), and a bunch of partitions. Traditionally, the first partition has something. Each partition will contain:

- Boot Entry
- FAT Information
- Optional redundant FAT
- Root Directory
- Datablocks

During a quick format, they will usually only change the formatting information, and the data is still dumpable. A full format changes all to 0. In Linux, it is possible to write complete gibberish using a random file.

File data is allocated to a number of data blocks or clusters. In the File Allocation Table, each data block has one entry, and it is a linked list storing information such as if it is free, next block (if it is EOF), and if it is damaged. OS can hold some of this information on RAM, though most of it might be in disk. Each FAT entry contains:

- FREE code
- Block Number to next block
- EOF code
- BAD code

Directories are represented as a special type fo file, and root directories are stored in a special location while other directories are stored as data blocks. Each file or subdirectory is represented as a directory entry.

12.1.1 8-3 Naming System

8 bytes are used for the file name, and 3 bytes are used for the file extension. 1 Byte is used for attributes, and 2 bytes each are used for creation date and time (1980 to 2107, accurate to 2 seconds). 2 bytes are used for the first disk block index (Number of bits depends on FAT type such as FAT12, 16, and 32), and the rest of the file can be accessed via the linked list. 4 bytes are used to keep the file size in bytes, and the remaining (10) bytes is reserved. The first byte of the name may have special meaning (DELTED, EOF, DIR)

Many things in FAT are kept for backward compatibility. The disk block numbers are used to perform actual disk access to allow for faster random access.

12.1.2 Variants

The main motivation is to be able to support larger hard disks. Disk clusters can be used (using a contiguous amount of disk blocks instead of a single as the smallest allocation unit) to store more data per unit / cluster. Increasing the size of the FAT also means there is are larger disk blocks and cluster, meaning more bits are needed to address each location.

In theory, a FAT12 with 4KiB clusters can have a largest partition (aka total space) of $4 \times 2^{10} \times 2^{12} = 16MiB$. FAT16 can have 256MiB, and FAT32 has 1TiB (uses only 2^{28} clusters as 4 bits are used for “future use”. In practice, clusters may not be fully utilized, may contain EOF, BAD, FREE, and information. Increasing the cluster size increases the usable partition, but may result in larger internal fragmentation.

12.1.3 VFAT

Virtual FAT allows filenames up to 255 characters. It is a workaround using multiple directories for a file with a long file name. There will still be a 8-3 short version for backward compatibility.

12.2 Linux Ext2 File System

In Extended2, it contains:

- BOOT
- Block Group(s)

And each block group contains information about itself:

- Data blocks
- Block bitmap to manage free space
- I-node table
  - Mode
  - Owner Info
  - File Size
  - Timestamps
  - Data Block Pointers (can be stacked to be direct or indirect pointers. There are 15 disk block pointers, of which 12 are direct blocks and the 13th is a single indirect block. 14 is a doubly indirect, and 15 is a triply indirect. This results in fast access to a small file, but flexibility in handling huge files.)
  - Reference Count
  - Other Fields
- I-node bitmap (meaning all I-nodes are same size)
- Group Descriptors (optionally duplicated)
  - Number of free disk blocks, free I-nodes, location of bitmaps in each block group.
- Super-block
  - Describes the whole file system
  - Total number of I-nodes, Disk Blocks
  - Duplicated in each block for each group

Unlike FAT, blocks can be stored in any position, as all its information is found locally.

Directories contain an INO number, the offset (pointer)to the next dirent, the length of the filename, the filetype, and the filename itself. For example, to access a file in /sub/file:

- Get I-node for root / directory (based on oS) (Thus it is always good to cache this)

- Get actual disk location from I-node to get another I-node index
- Use Inode index to get the I-node for sub
- Go to actual disk location to get the I-node index for file
- Use Inode index to get I-node to actual file
- Access the actual file.

12.2.1 File Deletion

When trying to delete /sub/file, need to check if information can be deleted (Group descriptors, bitmaps, I-node table). Then this information needs to be modified too.

12.2.2 Links

Hard links will point to the actual I-node of the data. The reference count will increase for that I-node. A soft link will have a another I-node pointing to the pathname. The link can be easily invalidated.