50004 - Operating Systems - Lecture 1

Oliver Killane

12/10/21

Lecture Recording

Lecture recording is available here

Introduction

Covering the theoretical side of OS, with many concepts to be implemented in the Pintos lab.

Course Delivery

Week	Form
2-3	Remote live lectures on teams (Tues and Thurs)
4	No lectures
5-11	Hybrid lectures (Teams & Huxley)

Part 1 - Dr Pietzuch

- Overview and Introduction
- Processes and Threads
- Overview and Introduction

Part 2 - Luis

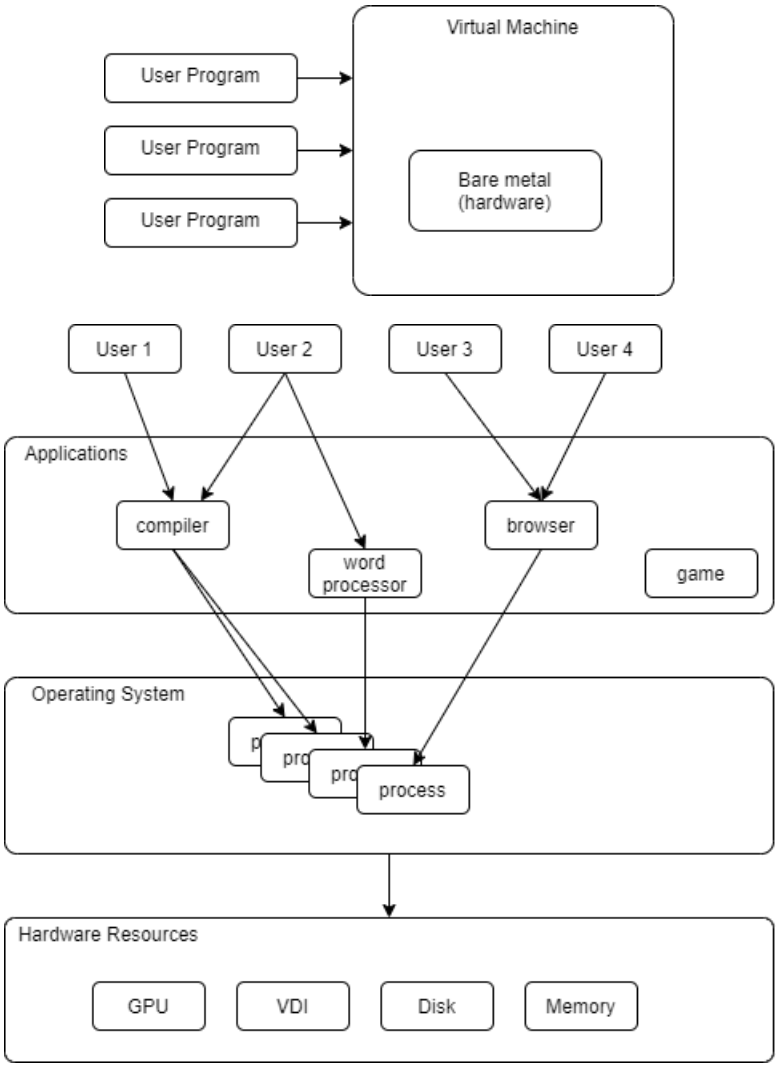
- Memory Management
- Device Management
- Disk Management
- File Systems
- Security
- Virtualisation

Recommended Reading

1. Modern Operating Systems
2. Operating Systems: 3 easy pieces (free online)
3. Understanding the Linux Kernel
4. Inside Windows NT & 2000

Computer Architecture Overview

OS provides a clean interface for different types of applications to run on different hardware.
OS provides resource management and useful abstractions.



OS Characteristics

The OS must be able to:

- Must share Data, programs and hardware (time and space multiplexing)
- Must offer resource allocation (efficient and fair use of memory, CPU etc)
- Must offer simultaneous access to resources, or if this is not possible, enforce mutual exclusion.

- Must protect against accidental or malicious corruption of Data.

Can also support concurrency (logical or actual)

- Support simultaneous activities occurring, e.g multiple users & programs
- Can switch activities at any arbitrary time, and must ensure this does not result in program failures
- Ensure safe concurrency through primitives (e.g semaphores, locks)

The OS must also take into account non-determinism, as it cannot predict what interrupts/events will occur and when. (events occur in an unpredictable order)

Also needs to consider persistent storage (e.g HDD, SSD etc).

- easy access to files through user-defined names.
- Enforces access controls (permissions to read, write, copy, etc).
- Can Protect against failure (e.g backups).
- Manage storage devices, partitions etc.

OS Structure

Monolithic

Whole OS is a single executable that runs in a single address space. File system, drivers etc are built into the kernel and run with privilege.

Microkernel

- Kernel provides some abstraction, manages resources, but many services run as servers (e.g file system, drivers, etc).
- IPC slow, so lower performance than monolithic.
- More reliable.

Hybrid Kernel

Mix of the two above, used by OS's such as windows.

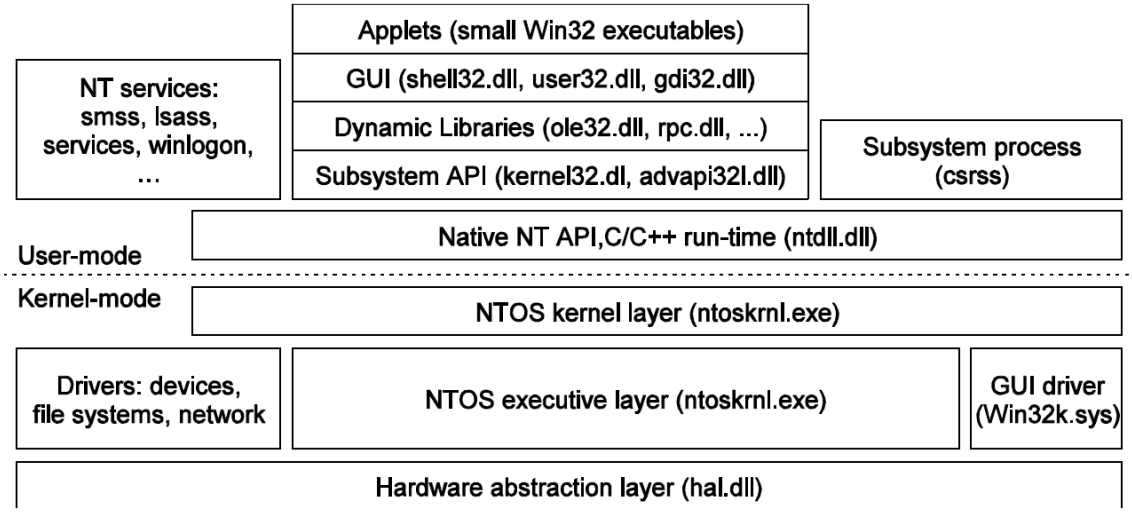
Linux

Linux is a variant of Unix, developed as a monolithic version of the Minix kernel.

- System calls implemented by putting args in registers & on the stack and issuing a trap.
- Supports many programs through GNU (GNU's not unix) project.
- Interrupt handlers are the primary method of interacting with devices.
- Supports dynamically loadable modules.
- exposes many devices and services as files.

Windows

A hybrid kernel, NT replaced MS-DOS and was inspired by VMS.



NTOS is loaded from ntoskrnl.exe (windows NT operating system kernel executable) at boot. It consists of two layers:

1. Executive - Most of the services.
2. Kernel Thread scheduling & synchronisation, traps, interrupt handlers, and CPU management.

There is also a **HAL** (Hardware Abstraction Layer) that abstracts away DMA (direct memory access) operations, BIOS config & CPU types. This in theory makes windows easy to poert to new devices and architectures. However in practice for mainly commercial reasons this has not occurred.

50004 - Operating Systems - Lecture 2

Oliver Killane

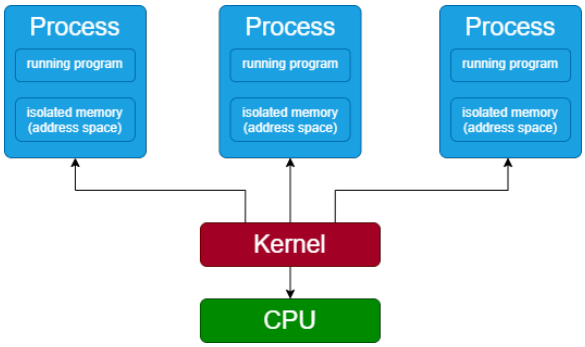
15/10/21

Lecture Recording

Lecture recording is available here

Processes

Processes are one of the oldest abstractions in computing, holding an instance of a running program.



The process abstraction can allow and OS to run programs simultaneously, even on only one CPU core.

Each process runs on a isolated **virtual CPU**, to achieve this the CPU resources of the system are multiplexed & managed by the OS.

Why Have Processes?

- **Provide Concurrency** To ensure real concurrency works properly (programs running on multiple CPU cores).

  And to manage a single CPU core so that multiple processes can appear to run simultaneously, each process running independently.

- **Provide Isolation** Each process has its own address space, and does not need to consider other unrelated processes in order to run correctly.

- **Simplify Programming** Applications can be easily developed to run without needing to consider how resources will be managed, or what other applications may be running. Programs can be written as if they will be run without any other programs running.

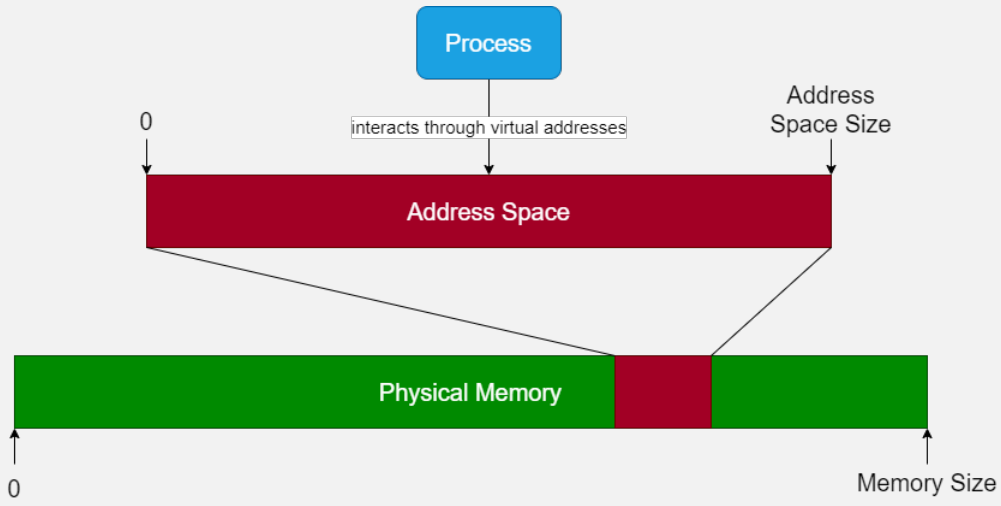
- **Better Resource Utilisation** Different processes require different resources. Hence we can more effectively increase utilisation & performance with multiple processes even on a single CPU core.

  For example a program may need to wait on I/O, rather than leaving this process idling while waiting, other programs can be scheduled to make use of resources during thwe wait.

Address Space

A process' address space is a contiguous region of memory allocated to a process by the operating system for use in storing data.

A process accesses this region using **virtual addresses** indexed from 0.



Multitasking

Time Slicing

Processes are given a set **time slice** to run on the CPU, before a **context-switch** is done to another process.

By switching between processes very quickly, the illusion of concurrency can be achieved on a single CPU core.

Context Switch

A context switch is where the CPU switches from the running one process/thread, to running another.

To achieve this the switch must:

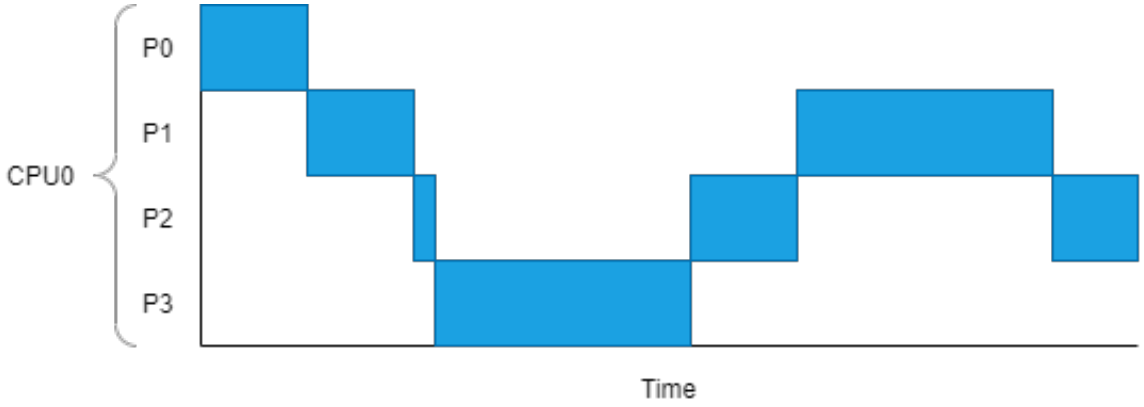
1. store the current state (Registers such as the stack pointer) such that it can be resumed later.
2. Load the previously stored state of another thread/process (registers such as stack pointer)
3. Resume processing the current thread (now switched)

This can occur due to an interrupt (e.g timer interrupts for a time slice strategy, accessing disk and other I/O etc) or in some systems when moving between **user mode** and **kernel mode** tasks.

Context switches are an essential feature of any multitasking operating system. Too many context switches can have a negative impact on performance.

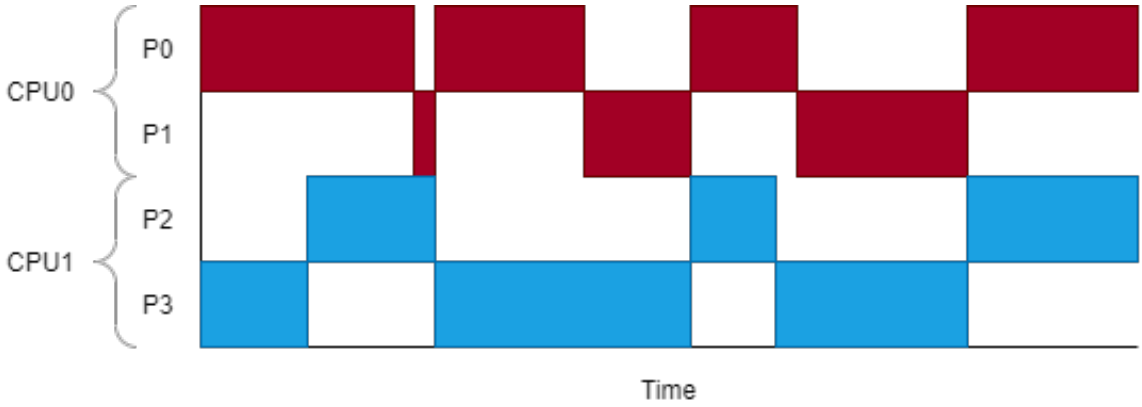
Pesudo Concurrency

Single processor, processes are interleaved to give the illusion of concurrency.



Real Concurrency

Several processors, running threads in parallel. Pesudo concurrency can also be employed (e.g in the diagram, 2 CPUs, 4 processes).



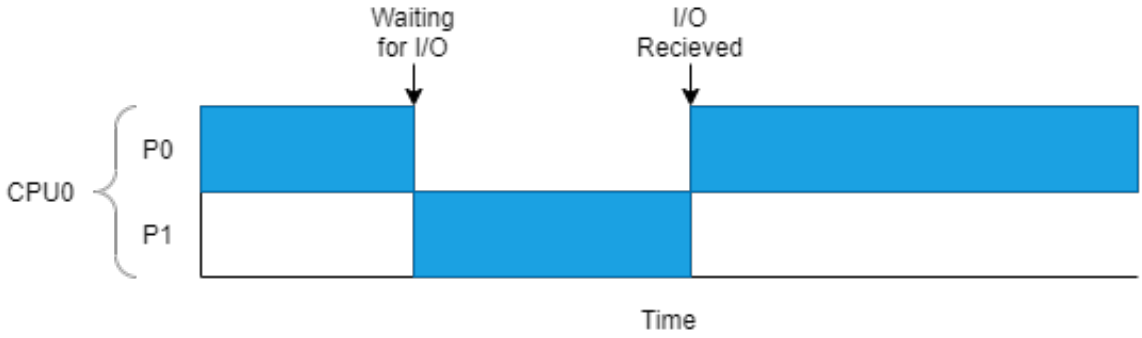
Fairness

When implementing a scheduler to determine when threads run we must consider how much time is allocated to each.

When one application is using too much CPU time, other processes may lagg (latency) and perform badly. Likewise if a process is not allocated sufficient resources.

Higher CPU Utilisation

We can switch to other threads when the current thread is waiting, instead of idling. This is beneficial so long as the context switch overhead is smaller than the utilisation that would've been lost to waiting.



We can compute an estimate for CPU utilisation on a uniprocessor:

$$\begin{array}{rcl} n & = & \text{total number of processes} \\ p & = & \text{Fraction of time a process is waiting for I/O} \end{array}$$

$$Prob(\text{all waiting for I/O}) = p^n$$

$$CPUutilisation = 1 - p^n$$

Process Control Block

A **Process Control Block** contains a process's information:

- **Process Identification** The process's unique ID, its group, parent etc. Used to identify what resources (e.g perhiperals) the process is using.
- **Process State Data** Values of saved registers (when suspended), such as the stack pointer, page table register etc.
- **Process Control Information**
  - Scheduling state (e.g ready, suspended, blocked)
  - Priority values (for priority scheduling)
  - Related process IDs (e.g parent)
  - **IPC** information (flags, signals and messages associated with communication among independent processes)
  - Process privileges (e.g file access, peripheral access)
  - Usage Statistics such as recent CPU time, used for scheduler priority calculations.
- **File Management Info** Root directory, working directory and open file descriptors.

Context Switch Expense

Context switches are expensive, and have considerable overhead, so we must avoid them if possible.

Direct Cost: save/restore process state.
Indirect Cost: pertubation of memory caches, TLB

Transaltion Lookaside Buffers

A cache storing recent translations of virtual addresses to physical addresses (used as user programs use virtual addresses to access their own address space).

It is a hardware feature that is part of a memory-management unit (MMU) and can reside between the CPU & CPU cahce, or between CPU cahce and the main memory.

During a context switch these are typically flushed.

Process Lifecycle

Process Creation

- System initialisation (start services, e.g file system)
- User request (run a program)
- Process Request (process makes a system call to start a new process)

Daemon

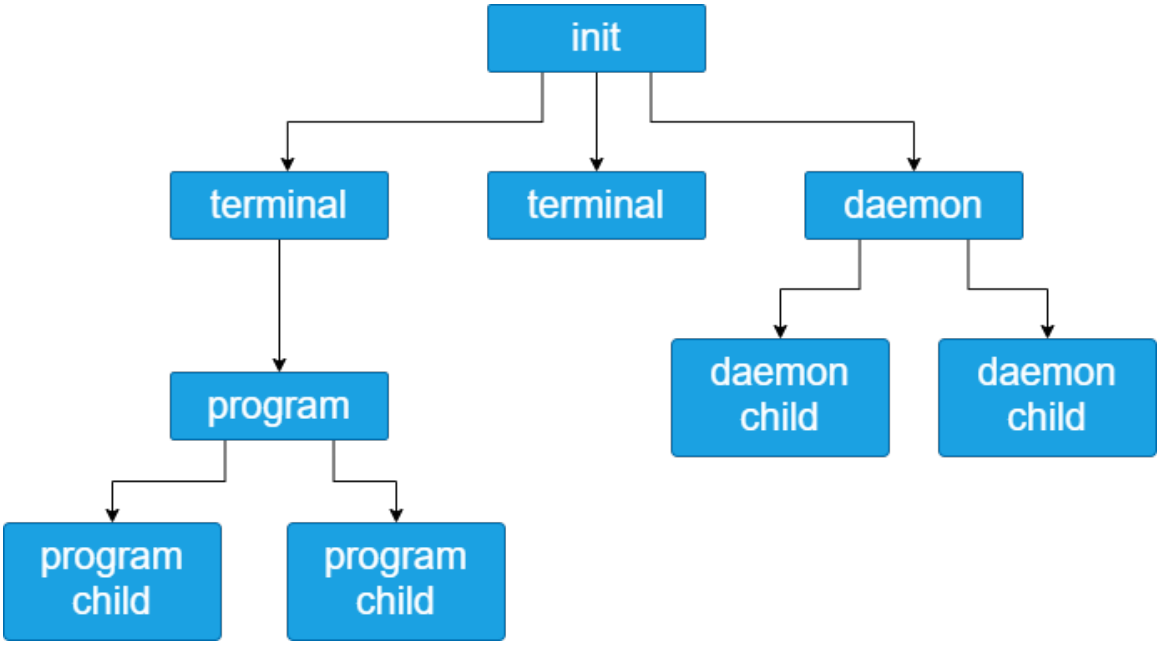
A background process, typically does not interact directly with users, e.g sshd (linux) listens for connections from clients, and forks a new daemon to manage each new connection.

Destruction

- **Completion**
  Once at the end of execution body (main function returns), the process ends.
- **System Call** **UNIX**: exit()
  Windows: ExitProcess()
- **Abnormal Exit**
  The process runs into an error or unhandled exception, and is forced to stop.
- **Aborted**
  Process stops because another process forces its execution to end. e.g parent process (terminal) forces child (program) to end.
- **Never**
  Some never terminate, many daemons need to be running at all timnes to ensure the OS or some crtitcal application being run functions as it should.

Process Hierarchies

UNIX has a strict tree of processes.



- On boot, **init** is created, this is the root of the process hierarchy tree (**process group**).

- Reads a file to determine how many processes need to be run, forks off one terminal per terminal.
- Each terminal waits for a user to login.
- On successful login, the login process executes a shell to accept commands which may in turn spawn more processes.

Windows has no such enforced hierarchy, upon spawning a process the parent is given a token/handle to control the child. This token can be transferred to other processes.

UNIX Processes

fork

```
1 int fork(void)
```

Fork creates an identical copy of the process, the child process iherits resources of the parent process and is run concurrently.

Return	Reason
0	You are the child
-1	fork failed (out of memory, max process limit reached, etc)
n	You are the parent, pid n is the child.

```
1  /* unistd provides access to the POSIX API */
2  #include <unistd.h>
3
4  /* fork is part of unistd */
5  int fork (void);
6
7  int main() {
8      int child_pid = fork();
9
10     if (child_pid != 0) {
11         /* parent code here */
12     } else {
13         /* child code here */
14     }
15 }
```

Executing Processes

```
1 int execve(const char *path, char *const argv[], char *const envp[])
```

Changes the process image and runs new process. Arguments:

- **path** Full pathname to program to run
- **argv** arguments to pass to main
- **envp** environment variables (e.g **$PATH) $HOME**

There are many useful wrappers to make calls easier/cleaner such as; execl, execle, execvp, execv...

Waiting for Process Termination

```
int waitpid(int pid, int* stat, int options)
```

Suspends the current process until the process **PID** has terminated.

PID	Result
-1	Wait for any child
0	Wait for any child in the same **process group** as the caller
$-gid$	Wait for any child with **process group** *gid*
pid	wait for process *pid*

> **Process Group**
>
> A collections of processes, typically used for signal distribution.
>
> When a **process group** is signalled, every contained process receives the signal, and can further direct it to other process groups.
>
> For example in the terminal, keyboard input can be sent to the terminal application, and the propagated to all running programs of that terminal instance.

Return value is as follows:

Value	Result
0	if **WNOHANG** is in set options (call should not block)
-1	if an error occured, errorno set to indicate error
pid	**PID** of process that terminated.

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

int main (int argc, char** argv) {

    char* command;
    char** parameters;

    for (;;) {
        get_command(&command, &parameters);

        if (fork()) {
            waitpid(-1, &status, 0);
        } else {
            execve(command, parameters, NULL);
        }
    }

}
```

Process Termination

```
void exit(int status)
```

Terminates the process (called implicitly when execution finishes), returns exit status to the parent process.

```
void kill(int pid, int sig)
```

Kill process using **PID**, by sending a signal to the process.

UNIX Philosophy

UNIX places value on building basic blocks to be combined into more complex programs, fork is an example of this, while execve could be used to replicate fork, it would be far more complex.

Windows does not follow this design philosophy, createProcess is the equivalent if fork and execve. It has 10 parameters for:

- program to execute
- parameters
- security attributes
- meta data for files
- priority
- pointer to info regarding new process
- and more!

```
BOOL WINAPI CreateProcess(
    __in_opt LPCTSTR lpApplicationName,
    __inout_opt LPTSTR lpCommandLine,
    __in_opt LPSECURITY_ATTRIBUTES lpProcessAttributes,
    __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,
    __in BOOL bInheritHandles,
    __in DWORD dwCreationFlags,
    __in_opt LPVOID lpEnvironment,
    __in_opt LPCTSTR lpCurrentDirectory,
    __in LPSTARTUPINFO lpStartupInfo,
    __out LPPROCESS_INFORMATION lpProcessInformation )
int pipe(int fd[2])
```

Process Communication

UNIX Signals

A limited form of **IPC** that works similarly to handler interrupts. A process can send a signal to another process if it has permission (same user for receiver or elevated privileges), the kernel can send a signal to any thread.

Signals can be triggered by:

- **Exceptions** (e.g Zero division error (SIGFPE) or segfault (SIGSEGV))

- **Events** e.g Kernel notifies a process if it writes to a closed pipe (SIGPIPE), or input (keyboard interrupt → SIGINT)
- **Programatically** Using the kill system call.

Common Signals are as follows

Signal	Description
SIGINT	Interrupt from keyboard
SIGABRT	Abort signal from abort
SIGFPE	Floating point exception
SIGKILL	Kill signal
SIGSEGV	Invalid memory reference
SIGPIPE	Broken pipe: write to pipe with no readers
SIGALRM	Timer signal from alarm
SIGTERM	Termination signal

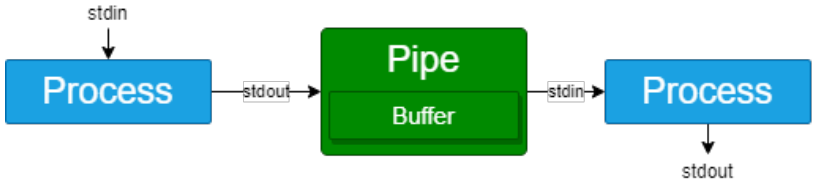
Signal Handlers

The default action for most signals is to terminate the process, however processes can register their own signal handlers.

```c
#include <signal.h>
#include <stdio.h>

void int_handler(int sig) {
    printf("SIGINT occured, but I'm ignoring it!");
}

int main(int argc, char** argv) {

    /* register handler */
    signal(SIGINT, int_handler);
    for(;;);
}
```

UNIX Pipes

A pipe connects the standard output of one process to the standard input of another.




```c
int pipe(int fd[2])
```

fd[0] read end (closed by sender)
fd[1] write end (closed by receiver)

When the receiver reads from an empty pipe, it is blocked until more data is written.
When the sender writes to a full pipe, it is blocked until data is read and removed from the pipe.

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

int main(int argc, char **argv) {

    /* pipe and buffer declaration */
    int fd[2];
    char buff;

    /* assert we have one argument string */
    assert(argc == 2);

    /* initialise pipe, if an error, return */
    if (pipe(fd) == -1)
        exit(EXIT_FAILURE);

    /* fork into parent and child */
    if (fork() != 0) {

        /* Parent: Sender */

        /* Close reading end (sender does not use this) */
        close(fd[0]);

        /* write the argument string to the pipe*/
        write(fd[1], argv[1], strlen(argv[1]));

        /* finished writing, now close write end */
        close(fd[1]);

        /* wait for the child to die*/
        waitpid(-1, NULL, 0);

        /* celebratory message */
        printf("Child Dead & Pipe Closed!\n");
    } else {

        /* Child: Receiver */

        /* close writing end (receiver does not use this) */
        close(fd[1]);

        /* read from pipe one character at a time, printing each char */
        while (read(fd[0], &buff, 1) > 0)
            printf("%c", buff);

        printf("\n");

        /* finished reading, now close the reading end */
        close(fd[0]);

        /* child reaches end of execution, terminates */
    }
}
```

UNIX Named Pipes

Persistent pipes outlive the processes which create them and are stored in the file system.

```
1 # create the pipe
2 mkfifo /some/pipe
3
4 # write to the pipe
5 echo "This is some text" >/some/pipe
6
7 # get text from the pipe
8 cat /some/pipe
```

50004 - Operating Systems - Lecture 3

Oliver Killane

21/10/21

Lecture Recording

Lecture recording is available here

Threads

Threads are execution streams that share the same address space. When multi-threading is used, each process can have more than one thread.

Process	**Thread**
Address Space	Program Counter
Global Variables	Registers
Open files	Stack
Child Processes	
Signals	

Applications may want to run many activities in parallel, with access shared memory. They also need to be capable of synchronisation, through semaphores, locks & monitors or blocking to prevent race conditions on shared data.

Processes have several disadvantages:

- **Communication**
  Difficult to communicate between different address spaces, requiring system calls (which have overhead if a context switch is required).

  If we split our program into several processes, each has a different address space, so all communication (internally in our program) has to use expensive system calls.
- **Blocking**
  Blocking activities may block the whole process, rather than one thread of a process.
- **Context Switches**
  Transfering between processes requires expensive context switches.
- **Create & Kill**
  Creation and desruction costs are higher.

Threads are a more lightweight abstraction for concurrency.

However threads access shared data, which means race conditions & memory corruption can occur.

PThreads

```
1 #include <pthread.h>
2 #include <sys/types.h>
3
4 pthread_t thread;              /* thread handle */
5 pthread_attr_t thread_attrs;    /* thread attributes */
```

Defined by **IEEE** standard 1003.1c and implemented on most **UNIX** systems. A library that internally makes systems calls to enable threading.

```c
#include <pthread.h>
#include <sys/types.h>

pthread_t thread;               /* thread handle */
pthread_attr_t thread_attrs;    /* thread attributes */
```

Creating Threads

```c
int pthread_create (pthread_t *thread , const pthread_attr_t *attr ,void *(*
    ↪ start_routine )(void*), void *arg)
```

Creates a new thread stored in **thread**, returning 0 for a successful creation, or an error code for failure.

Arguments:

- **attr**
  specifies thread attributes, can be **NULL** for default (e.g minimum stack stack size, guard size, detached/joinable etc)
- **start_routine**
  C function the thread will start executing from, returning some pointer, and taking some pointer as arguments.
- **arg**
  Arguments to be passed to start routine, can be **NULL** if no arguments are to be passed.

Terminating Threads

```c
void pthread_exit (void *value_ptr)
```

Terminates the thread and makes **value_ptr** available to any successful join with the terminating thread.

Is called implicitly when the thread's start routine returns.

- Not called for the initial thread (that started main).
- If **main** terminates before other threads without calling **pthread_exit**, the entire process is terminated.
- If **pthread_exit** is called in **main** the process continues executing until the last thread terminates (or **exit** is called.).

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

#define NUM_THREADS 10;
```

```c

/* takes a void pointer, which contains the id*/
void *thread_func(void *id) {
    printf("This is thread: %d!\n", (int)id);
}

int main (int argc, char** argv) {
    /* array containing threads */
    pthread_t threads[NUM_THREADS];

    /* create each thread and start */
    for (int id = 0; id < NUM_THREADS; id++)
        pthread_create(&threads[id], NULL, thread_func, (void *)id);

    /* process continues until all threads have terminated */
    pthread_exit(NULL);
}
```

Yielding Thread

```c
int pthread_yield (void)
```

Releases the **CPU** to let another thread run. Returns 0 on success. In linux it always succeeds.

Joining Threads

```c
int pthread_join (pthread_t thread, void **value_ptr)
```

Blocks until the thread terminates. The value passed to **pthread_exit** is available on location referenced by **value_ptr** (can be **NULL**) for example an error value may be propagated.

```c
#include <pthread.h>
#include <stdio.h>

void *thread_1_work(void *id) {
    printf("Thread 1 is working");

    /* code for task goes here */

    printf("Thread 1 is done!");
}

void *thread_2_work(void *id) {
    printf("Thread 2 is working");

    /* code for task goes here */

    printf("Thread 2 is done!");
}

int main (int argc, char** argv) {
    /* array containing threads */
    pthread_t thread_1, thread_2;

    printf("Both threads ordered to work!");

    pthread_create(&thread_1, NULL, thread_1_work, NULL);
```

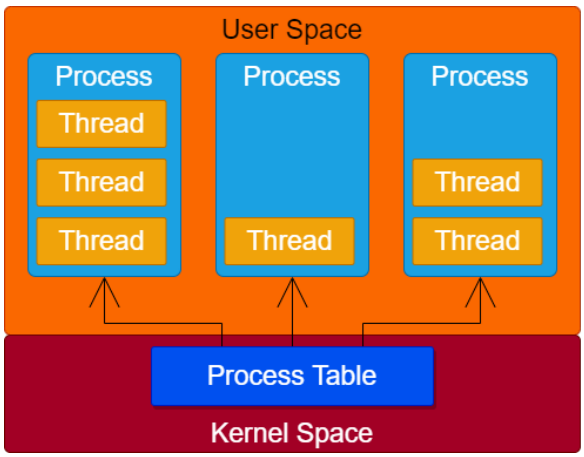
```
27      pthread_create(&thread_2, NULL, thread_1_work, NULL);
28
29      /* Join both threads to current thread. */
30      pthread_join(&thread_1, NULL);
31      pthread_join(&thread_2, NULL);
32
33      /* both have completed their work! */
34      printf("Both threads done working!");
35  }
```

OS Thread Implementation

The two main approaches are **User-Level Threads** and **Kernel-Level Threads**. Both have tradeoffs and hybrid approaches are possible.

User Level Threads

OS Kernel is manages processes only. Threads are implemented through a library, with running processes maintaining their own thread table to manage their threads.



Advantages

- **Better Performance**
  As the kernel is not involved in thread creation, termination, switching or synchronisation. They can be much faster.

  Fewer expensive context switches as less kernel involvement.

- **Optimisation**
  Each application can have its own scheduling algorithm, which can be optimised for the specific context of that application. Which aides performance.

Disadvantages

- **Blocking Calls** A blocking system call blocks the process (and all threads inside), which denies a key motivation for using threads.

While non-blocking I/O can be used (e.g **select**) they are more complex & inelegant.

- **Exceptions**
  Exceptions can block all threads, even if some are runnable (e.g page-fault).

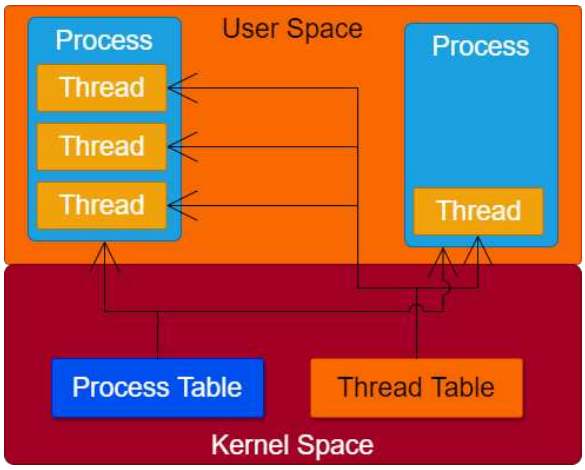
Select

select and **pselect** allow a program to monitor multiple file descriptors, waiting for some to become ready for some I/O before doing it.

```c
/* Example from tutorialspoint.com */
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    fd_set rfds;
    struct timeval tv;
    int retval;

    /* Watch stdin (fd 0) to see when it has input. */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /* Wait up to five seconds. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;
    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
        /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("No data within five seconds.\n");
    return 0;
}
```

Kernel Level Threads

The OS kernel manages the threads of processes directly.



Advantages

- **Blocking Calls**
  Blocking calls & exceptions (e.g page fault) do not block whole process, if one thread in a process is blocked, the kernel can schedule another runnable thread from that process.

Disadvantages

- **Thread Creation more expensive**
  Requires systemn calls (though is still cheaper than creating a new process).

  A mitigation stragetgy is to use thread pools (recycling threads).

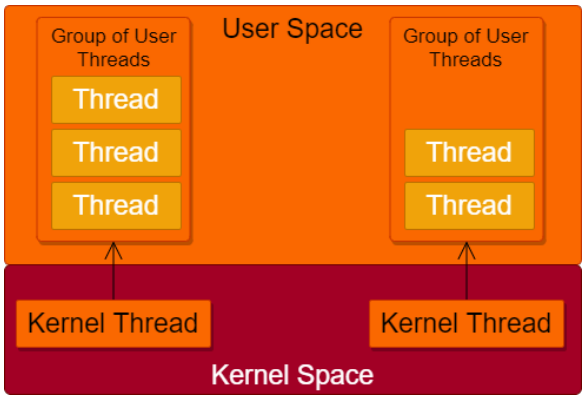
- **Synchronisation**
  Requires blocking system calls, which are expensive.

- **Switching more expensive**
  Requires a system call, kernel the schedule. Kernel scheduler be optimal for all applications (however still cheaper than switching processes).

- **No application-specific Schedulers!**

Hybrid Approaches

A suggested hybrid approach is to map user level threads, to a lower number of kernel threads. This has some of the advantages of both approaches, however requires some overhead in creating the mapping, and updating it.



The user level threads do not have 'real' concurrency in this approach, only kernel threads.

50004 - Operating Systems - Lecture 4

Oliver Killane

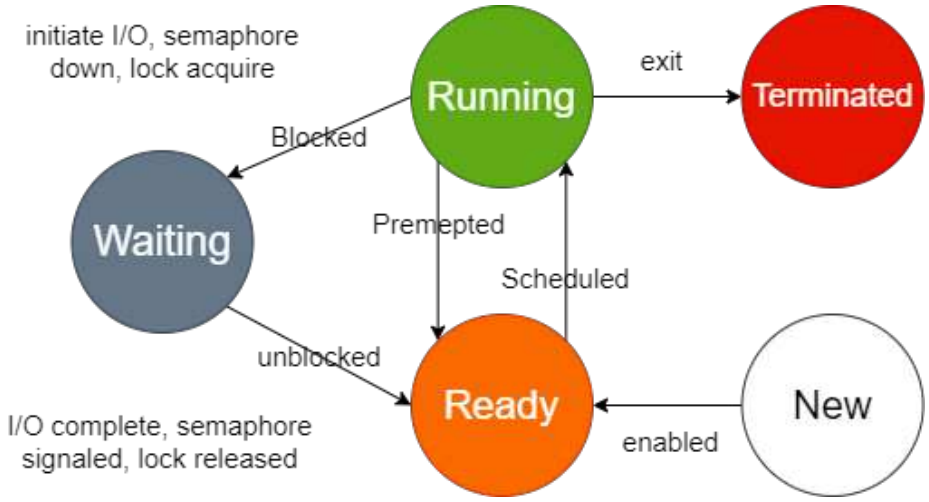
23/10/21

Lecture Recording

Lecture recording is available here

Scheduling

Process States



Scheduling concerns determining which ready process should be run, when and for how long.

Scheduling Goals

- **Fairness** Comparable processes get comparable resource allocation (e.g CPU time).
- **Avoid Indefinite Postponement** No process should starve (be left with no CPU time at all, effectively not running at all).
- **Enforce Policy** For example enforcing certain processes (e.g system processe) are prioritised.
- **Maximize Resource Utilisation** If there is a process that can be run, it should be run. Resources go beyond CPU time to resouces such as disk, or I/O.
- **Minimise Overhead** The scheduling system itself should use as little resources as possible. (e.g minimise scheduling decisions and frequency of context switches).

There are different priorities for different types of systems

Scheduler Types & Systems

- **Batch Systems** Optimise for throughput (jobs per unit time) or turnaround time (time from submission to completion).
- **Interactive Systems** Response Time is crucial (time from request issued to first response).

- **Real Time Systems** Need to meet deadlines. Soft: (recoverable) e.g reduced video quality. Hard: (irrecoverable) e.g factory robots collide.

There are two main types of scheduler

- **Non-Preemptive**
  - Processes are allowed to continue to run until they become blocked, or voluntarily yield the CPU.
  - Typically this requires software to be trusted to not monopolise the CPU maliciously.
  - Bad for interactivity as a process may be waiting for a long time.
  - Good for batch systems as sheduler overhead is typically small.
- **Preemeptive**
  - Processes can run for a maximum amount of time (time slice) before being descheduled.
  - Requires a clock interrupt to allow the OS to take control from the process.
  - Used often in interactive systems as longest wait is guarenteed to be (processes $-$ 1) $\times$ time slice.

CPU vs I/O Bound

CPU Bound

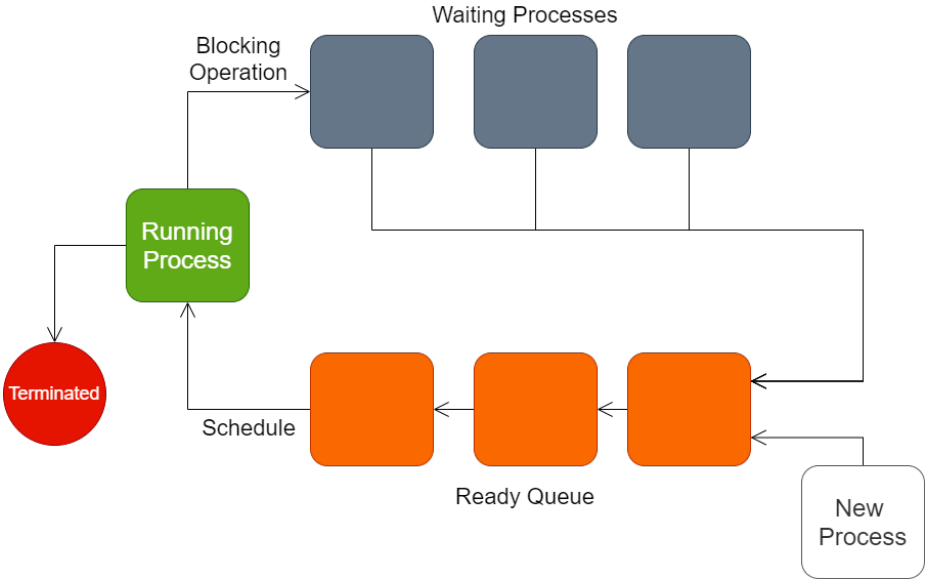
- Spend most of their time using the CPU (e.g data processing).

I/O Bound

- Spend most of their time waiting for I/O.
- Use the CPU briefly before issuing an I/O request.

Schedulers

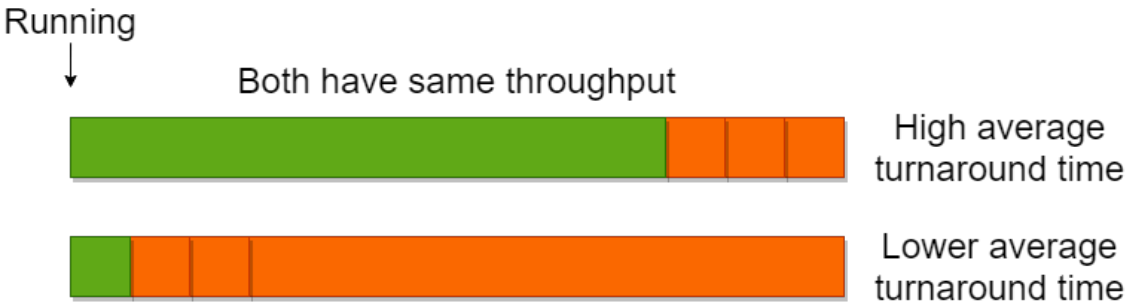
FCFS



Advantages

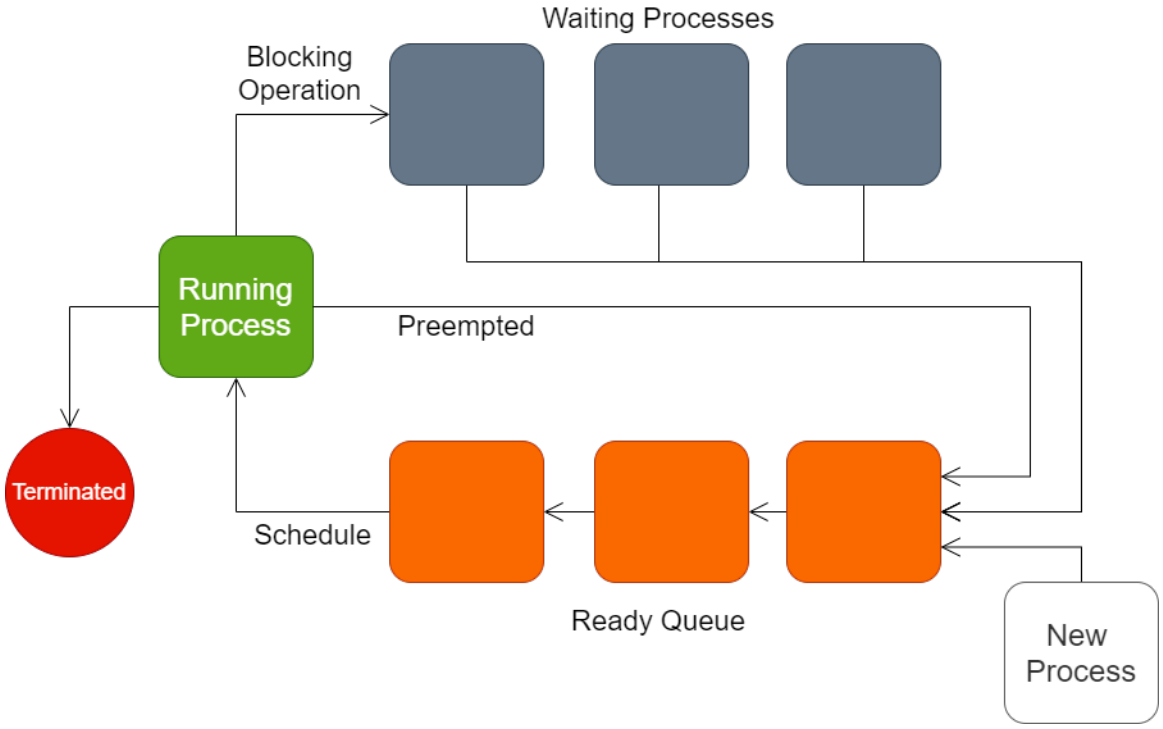
- All processes are eventually scheduled.
- Very simple & easy to implement.
- Very fast (minimal scheduling overhead).

Disadvantages
scheduler is unfair, for example:



Round Robin Scheduling

Much like **FCFS**, however a process is preempted and added back to the ready queue when its time quantum expires (time slice).



- Fair (ready jobs get equal CPU time).

- Response time is good for a small number of jobs.
- Turnaround time is low when run times differ (smaller jobs can finish quickly), but poor for similar runtimes (constant interruption & sent to back of ready queue).

$$\text{overhead} = \frac{\text{context switch}}{\text{context switch} + \text{quantum}}$$

$$\text{worst response} = \text{number of processes} \times \text{quantum}$$

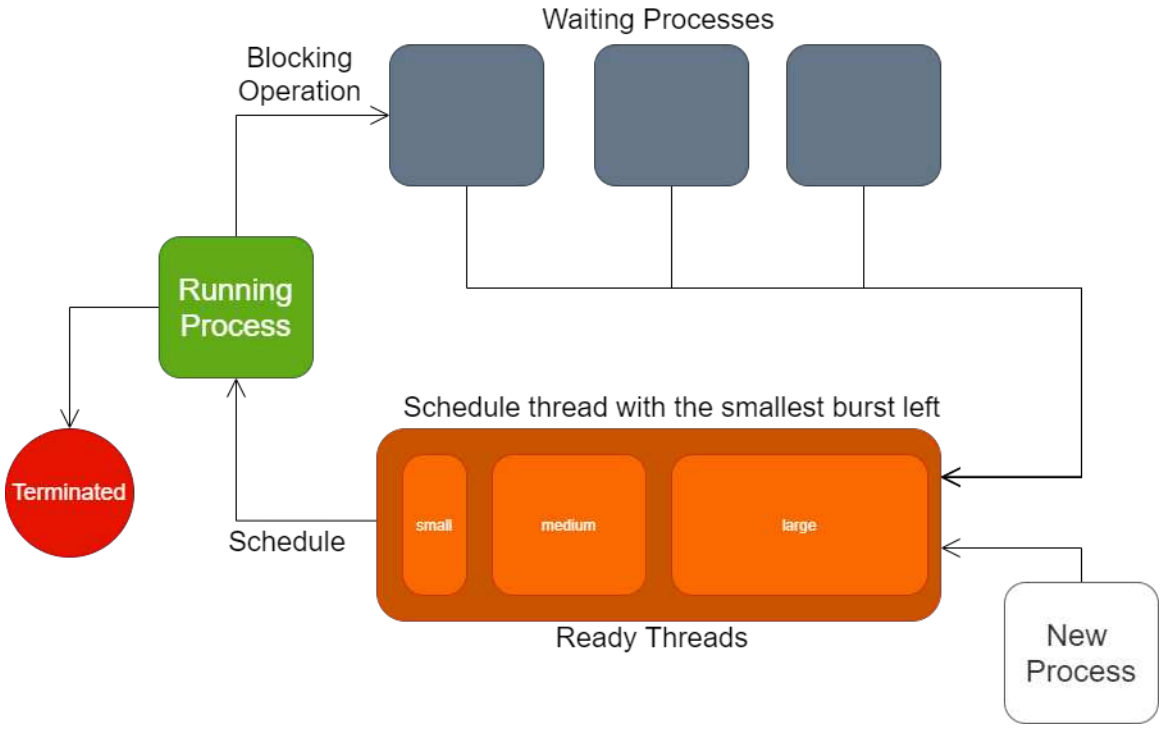
Hence the length of time quantum is a tradeoff between overhead and response time. It should be larger than the context switch time.

Typically from $10ms - 200ms$.

OS	**Time Slice**
Linux	$100ms$
Windows Client	$20ms$
WIndows Server	$180ms$

Shortest Job First

Schedule the ready thread with the shortest CPU burst remaining given we know the runtimes of each job (either provided, predicted or historical).



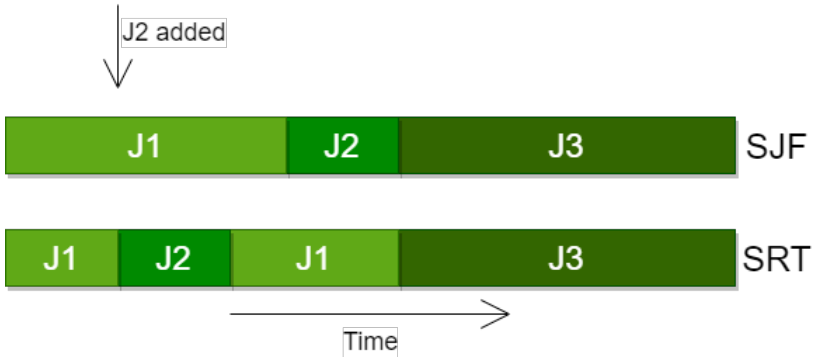
Average turnaround time is optimally low as we run the jobs that are quickest first.

Shortest Remaining Time

Premptive version of **SJF**, chooses the process with the shortest remaining time.

As **SJF** only makes scheduling decisions when a job is complete, when a new job is added while another job is running, it will have to wait. In **SRT** we can reschedule immediately upon receiving the shorter job. For example:

Job 1 & 2 added simultaneously, Job 3 added after $2s$:



One issue with **SRT** is that a long job could be heavily delayed if many short jobs are added.

Konowing Runtimes in Advance

Runtimes are normally not available in advance. However we can make estimates based on previous history (e.g past CPU time).

We can also use user estimates, however we may need to counteract programs cheating (by providing low estimates) by for example penalising processes that overrun their estimated time.

Fair Share Scheduling

On a multi-user system, we need to take into account which processes are being run by which users.

This can be achieved in **round robin** scheduling by having a ready queue per user, and round robining on which ready queue is used.

General Purpose Scheduling

- **Favour short & I/O Bound jobs**
  - Once scheduled they are quickly done with and can have resources freed.
  - I/O bound will likely be blocked and unscheduled quickly.
  - Keep reposnse times low.
- **Quickly Determine Job Nature**
  By quickly adapting to determine which jobs are **CPU** and **I/O** bound we can schedule more effectively.

Priority Scheduling

- Jobs run based on a priority. Always running the job with the highest priority.
- Priorities can be externally defined (user) or based on process-specific metrics (e.g expected CPU burst, previous CPU time, process parent).
- Can be static (unchanging) or dynamic (change during execution - e.g process aging).

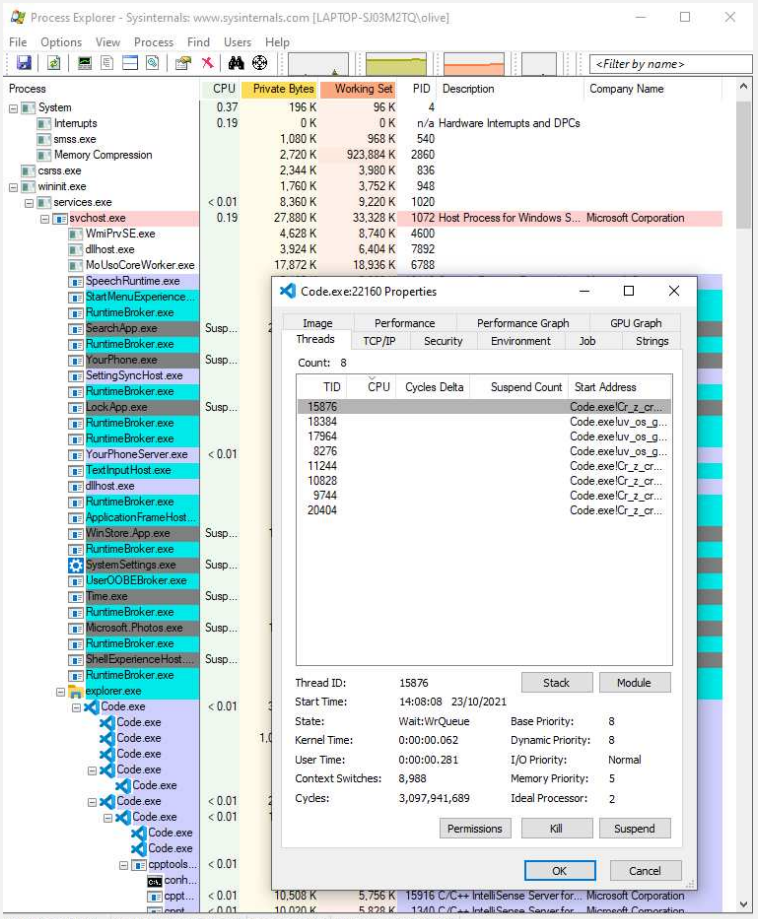
Note: Shortest remaining time is a form of priority scheduling.

These are also used in:

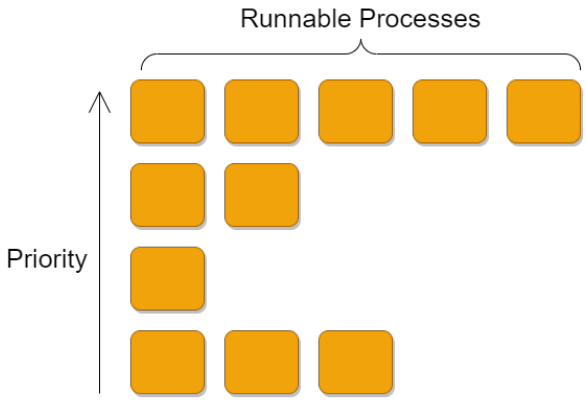
- Windows Vista, 7
- Mac OS X
- Linux 2.6 - 2.6.23
- Pintos!

Process explorer

If you are on windows, you can use the process explorer to see the priority system in action on any running process & its threads.
(https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer)



Multilevel Feedback queues



Each priority level has its own set of runnable threads, which can be scheduled (even by different algorithms).

We must be careful not to starve lower priority jobs. This can be done with the feedback mechanism that age a thread as it waits, and assign higher priority to threads that are waiting a long time.

We can also give I/O bound operations higher priority to ensure they are able to run as quickly they are done waiting (response time is important).

However there are some disadvantages:

- **Inflexible**
  Applications have no control over their priority, and priorities make no guarentees (e.g high priority does not matter, if all processes are running on high priority)
- **Cannot React Quickly**
  Often Needs a warm-up period (as initially priorities of processes may not relfect what bounds them/their behaviour). This also becomes a rpoblem for realtime systems that must react quickly.
- **Cheating**
  e.g Adding meaningless I/O to boost priority.

  As prity is based off 'feedback', priority can be manipulated.

Lottery Scheduling

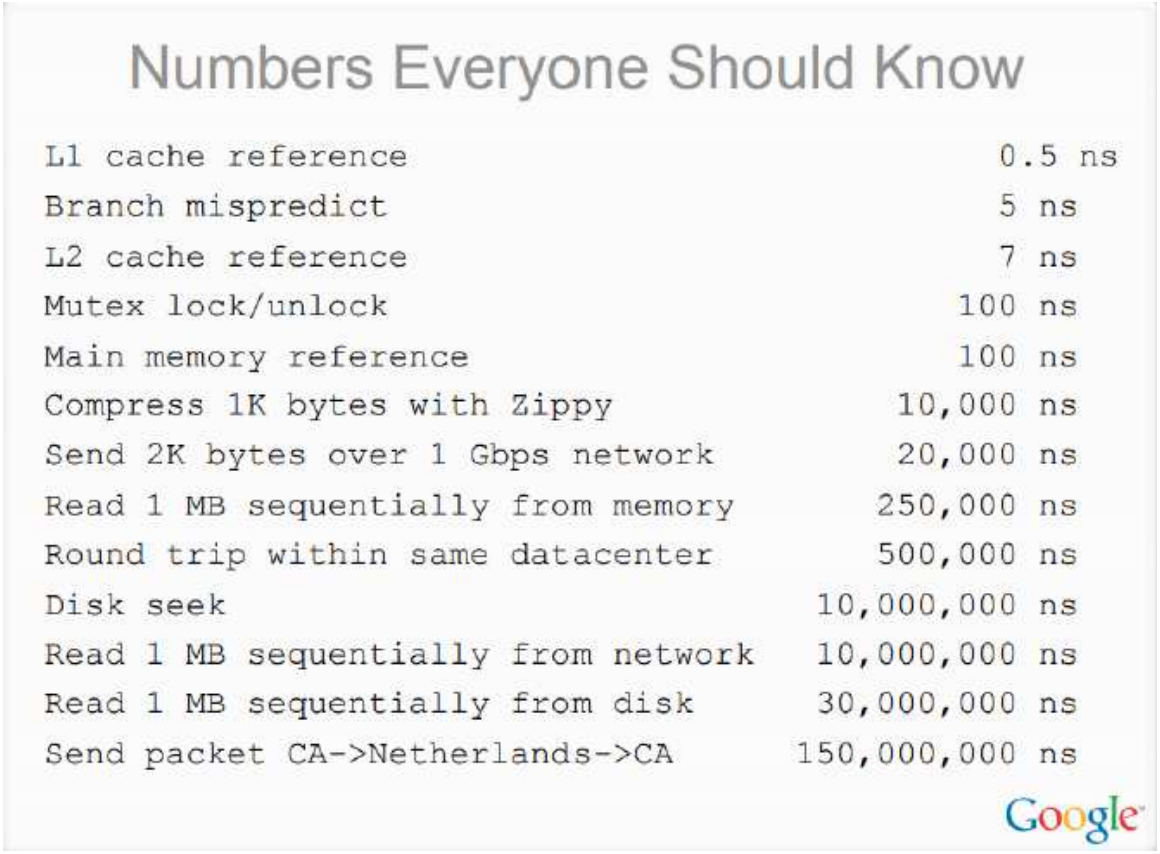
Next job to schedule is based on probability. At each scheduling decision a random process is chosen, with probability biased towards threads we that are higher priority.

- **tickets Meaningful**
  If a process has 20% of the tickets, it gets on average 20% of the time.
- **Highly responsive**
  We can provide a job more tickets to give it a higher chance of being run at the next scheduling decision.

- **No Starvation**
  As all threads have a chance of being scheduled at every decision.
- **Donation**
  Jobs can exchange tickets, this allows for easy priority donation, and allows cooperating jobs to achieve goals.
- **Adding a Job affects existing Jobs**
- **Unpredictable** A process can be *lucky* or *unlucky*, and this can impact response times (e.g interactive process is very unlucky, causes an issue).

Summary

- **Schedulers Balance Conflicting Goals** Fairness, enforce policy, maximise resource utilisation, minimise overhead.
- **Different Scheduling Algorithms are Appropriate in Different Contexts** Batch vs interactive vs real-time.
- **Well Studied Algorithms** FCFS, RR, SJF, SRT, MLFQs, Lottery



Additions

A little python program for doing **RR**, **fcfs** calculations:

```
from typing import List, Tuple

def fcfs(ps : List[Tuple[str,int]]):
    time = 0
    turnaround = 0
    for (p, t) in ps:
        time += t
        turnaround += t
        print("{} completed after {}".format(p, time))
    print("Avg turnaround: {}\nThroughput: {}", float(turnaround) / len(ps), float(
        ↪ time) / len(ps))

def rr(ps : List[Tuple[str,int]]):
    turnaround = 0
    ps = sorted(ps, key=lambda pt : pt[1])
    n = len(ps)
    lastt = ps[0][1]
    time = lastt * n
    print("{} completed after {}".format(ps[0][0], time))
    for (p, t) in ps[1:]:
        n -= 1
        time += (t - lastt) * n
        turnaround += time
        print("{} completed after {}".format(p, time))
        lastt = t
    print("Avg turnaround: {}\nThroughput: {}", float(turnaround) / len(ps), float(
        ↪ time) / len(ps))
```

50004 - Operating Systems - Lecture 5

Oliver Killane

02/11/21

Lecture Recording

Lecture recording is available here

Process Synchronisation

The key concepts to consider in process synchronisation are:

- **Critical Sections**
  Section of code where processes access a shared resource
- **Mutual Exclusion**
  Multiple threads/processes cannot execute in a critical section simultaneously.
- **Atomic Operations**
  Operations that can occur without interruption or interleve by other threads, e.g reading and writing in 1 instruction.
- **Race Conditions**
  Where the behaviour of the program is dependent on the timing and interleving of threads.
- **Deadlock**
  Where the completion of tasks are mutually dependent, there is some loop in the dependency graph meaning a task's completion cannot occur before its completion.
- **Starvation**
- **Synchronisation Mechanisms**
  Locks, Semaphores and Monitors for example.

  Primitive that allow programmers to enfoce ordering of tasks, and exclusion. They are required at entry to and exit from a critical section.

Requirements of Mutual Exclusion

- No two processes can simultaneously be in a critical section.
- No process running outside the critical section can prevent another from entering the critical section.
- When no process is in the critical region, any process requesting permission to enter must be immediately admitted.
- No process requiring access to a critical section can be delayed indefinitely.
- No assumptions are made about the relative speed of processes.

Disabling Interrupts

```
1 void Extract(int acc_no , int sum)
2 {
3     CLI();
4     int B = Acc[acc_no];
5     Acc[acc_no] = B - sum;
6     STI();
7 }
```

- Works only on single-core Systems (other core may access shared in memory).
- May never release the CPU if code buggy.
- Interrupts could be missed while interrupts disabled.
- Only possible in kernel code which has the ability to disable interrupts.

Software Solution - Strict Alternative

```
1 while (true) {
2     while (turn != 0)
3         /* loop */ ;
4     critical_section();
5     turn = 1;
6     noncritical_section0();
7 }
```

```
1 while (true) {
2     while (turn != 1)
3         /* loop */ ;
4     critical_section();
5     turn = 0;
6     noncritical_section0();
7 }
```

Issues:

- Cannot run 0 or 1 twice in a row without the other working.
- Busy waiting of thread on turn variable.
- If the noncritical section takes a long time, the other thread is blocked as it must wait for turn to be set. (Thread outside critical region prevent critical region access).
- Turn must be volatile to prevent compiler optimisations causing issues.

Busy Waiting

Continuously checking a value to determine if a thread can progress. It is a waste of CPU time and should only be used if the wait is short.

However as a result of actively polling the a value, it can stop waiting very quickly. Spinlocks used in OS kernels take advantage of this.

Peterson's Solution

```c
int turn = 0;
int interested[2] = {0, 0};

// thread is 0 or 1
void enter_critical(int thread) {
    int other = 1 - thread;
    interested[thread] = 1;
    turn = other;
    while (turn == other && interested[other])
        /* loop */ ;
}

void leave_critical int thread) {
    interested[thread] = 0;
}
```

```c
enter_critical(1);
critical_section();
leave_critical(1);
```

```c
enter_critical(0);
critical_section();
leave_critical(0);
```

This lock busy waits while the the other is interested and it is the other's turn.

Atomic Operations

```c
void Extract(int acc_no , int sum)
{
    int B = Acc[acc_no];
    Acc[acc_no] = B - sum;
}
```

Not atomic, other extract could be interleaved.

$\Rightarrow$

```c
void Extract(int acc_no , int sum)
{
    Acc[acc_no] -= sum;
}
```

Not atomic, still requires multiple instructions, between which another process could be interleaved.

Locks

```c
void Extract(int acc_no , int sum)
{
    lock(L);
    int B = Acc[acc_no];
    Acc[acc_no] = B - sum;
    unlock(L);
}
```

```c
void lock(int *L)
{
    while (L != 0)
        /* wait */;
    *L = 1;
}
```

```c
void unlock(int *L)
{
    *L = 0;
}
```

Here the implementation of the lock is flawed as we cannot atomically check the value of the lock, and set its value.

A working implementation is:

```c
void lock(int *L)
{
```

```
3       while(TSL(L) != 0)
4           /* wait */ ;
5   }
```

TSL (test-and-set-lock) atomically sets the given memory location to 1 and returns the old value. Locks of this type are called **spin locks**.

Spin Locks

Spin locks should only be used when the expected wait time is short (otherwise wasteful). However it may run into the priority inversion problem.

Priority Inversion

When a lower priority thread is scheduled instead of a higher priority thread.

Threads $H > M > L$.
L acquires a lock, however is descheduled. H attempts tpo acquire the lock, but is blocked. The lock is not released as L needs to run in order to release it. M is higher priority than L, and runs instead.

50004 - Operating Systems - Lecture 6

Oliver Killane

04/11/21

Lecture Recording

Lecture recording is available here

Lock Granularity

```
1 void Extract(int acc_no , int sum)
2 {
3     lock(L);
4     int B = Acc[acc_no];
5     Acc[acc_no] = B - sum;
6     unlock(L);
7 }
```

$\Rightarrow$

```
1 void Extract(int acc_no , int sum)
2 {
3     lock(L[acc_no]);
4     int B = Acc[acc_no];
5     Acc[acc_no] = B - sum;
6     unlock(L[acc_no]);
7 }
```

- **Lock Overhead** measure of cost associated with a Lock:
  Consider memory use, initialisation cost & the cost of acquiring and releasing locks.
- **Lock Contention** measure of the number of processes waiting on a lock:
  More contention $\rightarrow$ less parallelism

Tradeoffs between fine and coarse grained synchronisation are context dependent, however it is always beneficial to keep critical regions as small as possible.

```
1 void AddAccount(int acc_no, int balance)
2 {
3     lock(L_Acc);
4     CreateAccount(acc_no);
5     lock(L[acc_no]);
6     Acc[acc_no] = balance;
7     unlock(L[acc_no]);
8     unlock(L_Acc);
9 }
```

Read/Write Locks

For two threads accessing data concurrently we can have race conditions if at least one is changing the shared data.

		Thread A	
		read	write
Thread B	read	Safe	Race
	write	Race	Race

```
1 void ViewHistory(int acc_no)
2 {
3     lock_RD(L[acc_no]);
4     print_transactions(acc_no);
5     unlock(L[acc_no]);
6 }
```

```
1 void Extract(int acc_no, int sum)
2 {
3     lock_WR(L[acc_no]);
4     Acc[acc_no] -= sum;
5     add_debit(acc_no, sum);
6     unlock(L[acc_no]);
7 }
```

- **lock_RD** Acquire the lock in read mode, if another lock has write mode, blocks.
- **lock_WR** Acquire lock in write mode, if any other thread has the lock in read or write mode, blocks.

Memory Models

In the course we assume sequential consistency:

- Operations in each thread occur in the order expressed in code/program order.
- Operations in each thread are executed in sequential order atomically.
- Hence race we can reason more easily about the behaviour of a thread in isolation.

However there are other memory models (due to compiler optimisations that split up or re-order operations and hardware (e.g core cache coherence)).

```
1 /* shared data */
2 int a, b;
```

```
1 void A() {
2     a = 1;
3     b = 1;
4 }
```

```
1 void B() {
2     b = 2;
3     a = 2;
4 }
```

Note that we assume a single write is atomic, and immediately visible to threads running on other cpu core, this is typically not the case.

Most architectures attempt to be cache coherent (illusion of coherent memory across cores), there are cases where this coherence is not achieved.

$a = 1$	$a = 1$	$a = 1$	$b = 2$	$b = 2$	$b = 2$
$b = 1$	$b = 2$	$b = 2$	$a = 2$	$b = 1$	$b = 1$
$b = 2$	$b = 1$	$a = 2$	$a = 1$	$a = 2$	$b = 1$
$a = 2$	$a = 2$	$b = 1$	$b = 1$	$b = 1$	$a = 2$
$2, 2$	$2, 1$	$2, 1$	$1, 1$	$2, 1$	$2, 1$

```
1 /* shared data */
2 int flag1 = 0, flag2 = 0;
```

```
1 void A()
2 {
3     flag1 = 1;
4     if (flag2 == 0)
5         critical();
6 }
```

```
1 void B()
2 {
3     flag2 = 1;
4     if (flag1 == 0)
5         critical();
6 }
```

With sequential consistency, it is impossible for $flag1 == flag2 == 0$ however under a weak memory model, due to the if statements not being dependent on the assignments, they may be reordered such that this is possible.

Happens-Before Relationship

Formulated by Leslie Lamport in 1976. It is a partial order realtion between events in a trace denoted by $a \rightarrow b$ where a, b are events in a trace.

Partial Order

A partial order is:

- **Irreflexive** $\forall a.[a \not\rightarrow a]$
- **Antisymmetric** $\forall a, b.[a \rightarrow b \Rightarrow b \not\rightarrow a]$
- **Transitive** $\forall a, b, c.[a \rightarrow b \wedge b \rightarrow c \Rightarrow a \rightarrow c]$

Consider a and b in a trace where a occurs before b.

- if a, b are in the same thread, then $a \rightarrow b$.
- if a, b are not in the same thread and $a : unlock(L)$ and $b : lock(L)$ then $a \rightarrow b$ (cannot lock until it is unlocked by other thread.)

There are other orderings deducible from the synchronisation primitives. A **Race Condition** occurs between a and b iff:

- the access the same memory location
- at least one is a write
- they are unordered according to **Happens-Before**

```
1 /* shared data */
2 int a, b;
```

```
1 void A() {
2     a = 1;
3     b = 1;
4 }
```

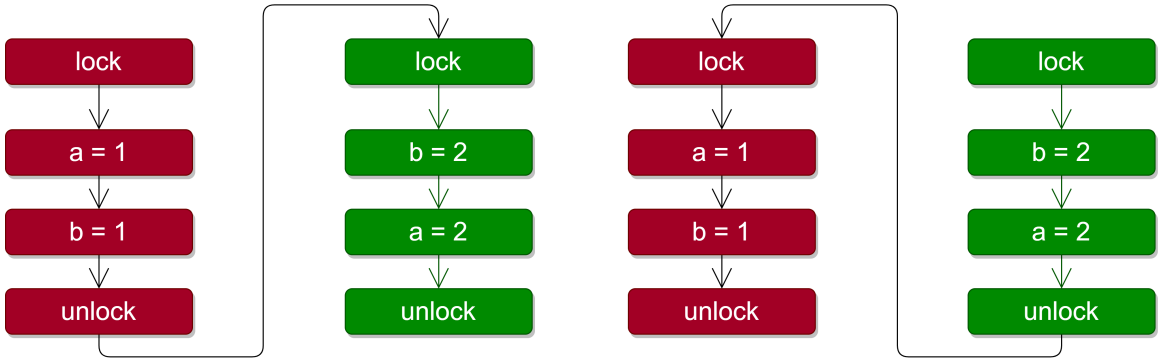
```
1 void B() {
2     b = 2;
3     a = 2;
4 }
```

Hence there is a data race between $a = 1, a = 2$ and $b = 2, b = 1$

```
1 void A() {
2     lock(L);
3     a = 1;
4     b = 1;
5     unlock(L);
6 }
```

```
1 void A() {
2     lock(L);
3     b = 2;
4     a = 2;
5     unlock(L);
6 }
```

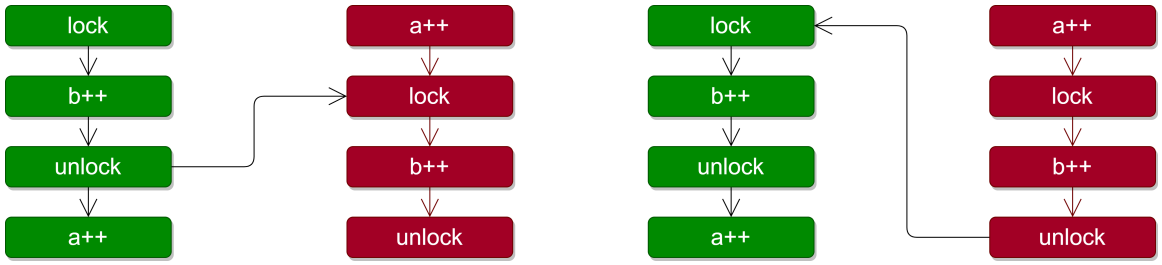
Two possible execution traces would be:



There are no race conditions in either.

```
1  void A() {
2      a++;
3      lock(L);
4      b++;
5      unlock(L);
6  }
```

```
1  void A() {
2      lock(L);
3      b++;
4      unlock(L);
5      a++;
6  }
```



Notice that while the right execution trace has no race condition, the left does.

50004 - Operating Systems - Lecture 7

Oliver Killane

09/11/21

Lecture Recording

Lecture recording is available here

Semaphores

A blocking synchronisation mechanism invented by **Dijkstra**. Processes cooperate by means of signals:

- Process stops, waiting for a signal.
- Process continues if it has received a specific signal.

Semaphores allow **i** processes to be between **down** and **up**. They are special variables containing a value and a queue of waiting processes, they are accessible through the following atomic operations:

- **down(s)** wait to receive signal via **semaphore s**.

```
1  void down(Sema *s)
2  {
3      if (s->counter > 0) {
4          s->counter--;
5      } else {
6          Process *cur = current_process()
7          queue_add(cur)
8          suspend_process(cur)
9      }
10
11 }
```

- **up(s)** transmit a signal via **semaphore s**.

```
1  void up(Sema *s)
2  {
3      if (queue_empty(s->waiters))
4          s->counter++;
5      else
6          resume_process(queue_pop(s->waiters))
7  }
```

- **init(s,i)** initialise semaphore **s** with value **i**

```
1  typedef struct {
2      int counter;
3      queue waiters;
4  } Sema;
5
6  void init(Sema *s, int i)
7  {
8      s->counter = i;
9      queue_init(s->waiters);
10 }
```

Mutual Exclusion

```
Sema s;

int main(int arc, char **argv)
{
    sema_init(&s, 1);

    /* Start processes. */
    ...
}


void some_function(void)
{
    sema_down(&s);

    /* Critical region. */

    sema_up(&s);
}
```

A semaphore with an initial value of 1 acts similarly to a **lock** (**lock** also enforces that only the thread that acquired the access to the critical region can release it).

Ordering Events

```
Sema s;

int main(int argc, char **argv)
{
    sema_init(&s, 0);
}
```

```
void process_A(void)
{
    ...
    /* Critical region */
    sema_up(&s);
}
```

```
void process_B(void)
{
    ...
    sema_down(&s);
    /* Critical region */
}
```

No matter the order of processes being started, process B cannot continue to the critical region until process A has called **sema_up**.

Producer/Consumer

- **Producer Constraints**
  - Items can only be deposited if there is space.
  - Items can only be deposited in the buffer if mutual exclusion is ensured.
- **Consumer Constraints**
  - Items can only be fetched if the buffer is not empty.
  - Items can only be fetched from the buffer if mutual exclusion is ensured.

- **Buffer Constraints**
  Buffer can hold between 0 and N items.

```c
/* Global semaphores and buffer size. */
Sema item, space, mutex;
int N;

int main(int argc, char **argv)
{
    sema_init(item, 0);  /* Number of items available, blocks consumer if zero. */
    sema_init(space, N); /* Remaining space available, block producer if zero. */
    sema_init(mutex, 1); /* Mutual exclusion access to buffer. */

    /* Create processes. */
    ...
}
```

```c
void producer(void)
{
    for (;;) {
        /* Produce item. */
        sema_down(space);
        sema_down(mutex);
        /* Desposit item. */
        sema_up(mutex);
        sema_up(item);
    }
}
```

```c
void consumer(void)
{
    for (;;) {
        sema_down(item);
        sema_down(mutex);
        /* Fetch item. */
        sema_up(mutex);
        sema_up(space);
        /* Consume item. */
    }
}
```

Monitors

Monitors allow processes to wait on high level conditions. Entry procedures are called from outside the monitor, internal procedures only from inside (cannot access monitor data, only monitor procedures).

There is an (implicit) monitor lock, with one process being in the monitor at a time.

The high-level conditions could be:

- *Some space is available in the buffer*
- *Some data has arrived in the buffer*
- *There is an available resource*

Signals do not accumulate, if a condition is signalled and no processes/threads are waiting for it, the signal is lost.

Typically a language construct, and are not supported by **C**, however we can explicitly implement them (as in **PINTOS**).
Java uses synchronize, **wait** and **notify** with no condition variables.

Monitor Procedures

- **wait(condition)** Releases monitor lock, waits for **condition** to be signaled.
- **signal(condition)** Wakes up one process waiting for **condition**.
- **broadcast(condition)** Wakes up all processes waiting for **condition**.

Monitor Code

```
1  montior ProducerConsumer {
2      condition not_full, not_empty;
3      int count = 0;
4
5      entry procedure insert(item) {
6          while (count == N) wait(not_full);
7          insert_item(item);
8          count++;
9          signal(not_empty);
10     }
11
12     entry procedure remove(item) {
13         while (count == 0) wait(not_empty);
14         remove_item(item);
15         count--;
16         signal(not_full);
17     }
18 }
```

Synchronisation Summary

- **Lock**
  - Reader/Writer Locks.
  - Often exposed in monitor language construct.
  - Within a process.
  - 1 process/thread in the critical section.
- **Mutex**
  - Like lock, but works across processes.
- **Semaphore**
  - Like mutex, but can let N processes / threads pass & does not need the same process/thread to release as acquired.

50004 - Operating Systems - Lecture 8

Oliver Killane

09/11/21

Lecture Recording

Lecture recording is available here

Lecture Recording

Lecture recording is available here

Deadlock

When two processes mutually block access to a resource the other requires to progress. An example is below:

```
Sema scanner, dvd_writer;

int main(int argc, char **argv)
{
    sema_init(&scanner, 1);
    sema_init(&dvd_writer, 1);
}
```

```
void process_A(void)
{
    sema_down(&scanner);
    sema_down(&dvd_writer);
    scand_and_record();
    sema_up(&dvd_writer);
    sema_up(&scanner);
}
```

```
void process_B(void)
{
    sema_down(&dvd_writer);
    sema_down(&scanner);
    scand_and_record();
    sema_up(&scanner);
    sema_up(&dvd_writer);
}
```

Dining Philosophers

```
#define PHILOSOPHERS 5

Sema chopsticks[PHILOSOPHERS];

int main(int argc, char **argv)
{
    /* Initialise chopstick semaphores. */
    for (int i = 0; i < PHILOSOPHERS; i++) {
        sema_init(&chopsticks[i], 1);
    }

    /* Start philosophers. */
    ...
}
```

```
void eat_and_think(int i)
{
    for (;;) {
        sema_down(&chopstick[i]);
        sema_down(&chopstick[(i + 1) % PHILOSOPHERS]);
```

```
6           /* Eat using chopsticks. */
7           sema_up(&chopstick[i]);
8           sema_up(&chopstick[(i + 1) % PHILOSOPHERS]);
9           /* Think about philosophy for a while. */
10      }
11  }
```

Notice that if every philosopher takes chopstick i at the same time (chopstick to the left), then we have a deadlock. As no one can get the chopstick to the right, so cannot progress.

A set of processes is **deadlocked** if each process is waiting for an event that only another process can cause, and that process is waiting on an event that is directly or indirectly caused by the other.

Resource deadlock is the most common type, and for it 4 conditions must hold:

- **Mutual Exlcusion** Each resoruce is either available or assigned to exactly one process
- **Hold & Wait** Process can request resources while it holds other resources.
- **No preemption** Resources given to a process cannot be forcibly revoked or borrowed.
- **Circular Wait** Two or more processes in a circular chain, each waiting for a resoruce held by the next process.

Resource Allocation Graphs

A **Directed Graph** containing:

A cycle in the graph represents a deadlock.

Deadlock Strategies

- **Ignore**
  The "ostrich algorithm", if contention for resources is low & deadlocks very infrequent, then can iugnore, simply restart when a deadlock occurs.
- **Detection & Recovery**
  After the system becomes deadlocked, detect a deadlock has occured & recover from it (e.g reverting to a saved state).
- **Dynamic Avoidance**
  Dynamically consider every request and decide if it is safe to grant.

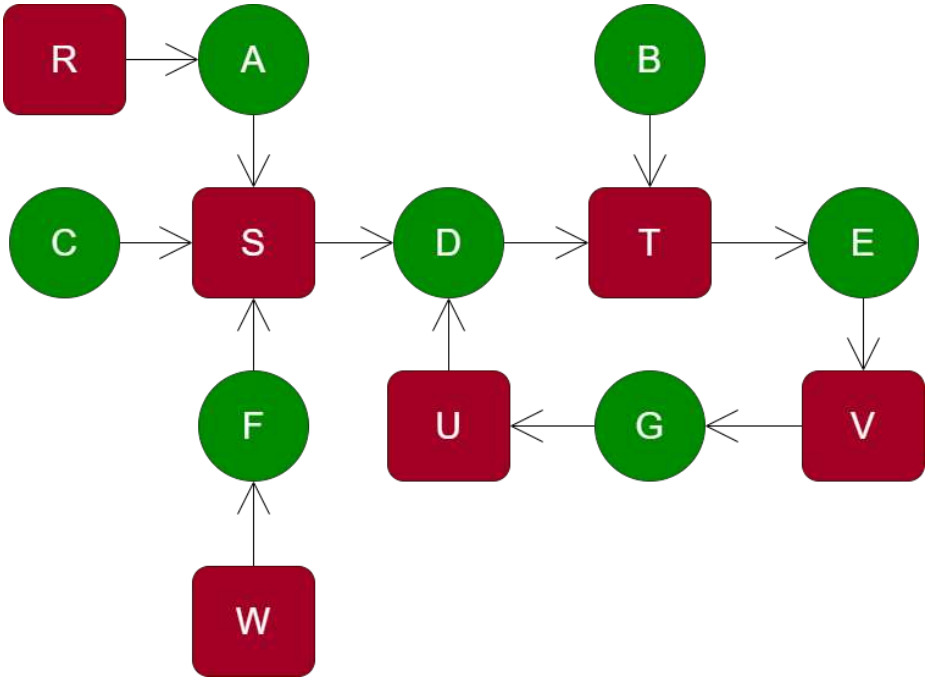
  Needs information to determine safety of a request (e.g resoruce use, current owners etc).
- **Prevention**
  Prevent deadlocks by ensuring at least one of the four conditios can never hold.

Detection & Recovery

Dynamically builds resource ownership graph and searches for cycles (**Depth-First Search**, when edge inspected it is marked and not visited again).

When cycle detected, recover.

```
# Code to detect a cycle in the resource allocation graph
def traverse(node, visited):
    if node in visited:
        CYCLE!!
    else:
        visited.append(node)
        for outgoing_edge in node:
            if outgoing_edge.unmarked:
                outgoing_edge.mark
                traverse(outgoing_edge.to, visited)

def detect(graph):
    for node in graph:
        traverse(node)
    NO CYCLE!!
```



Here we have a cycle as $D, E\&G$ are in a cycle and cannot gain the resources they need to progress. Further B may be blocked as E holds T.

Recovery

- **Pre-emption**
  Temporarily take a resource from an owner & give it to another.

- **Rollback**
  Processes are periodically checkpointed (memory image, state), rollback to the last state (note we must still prevent deadlock from occuring when this state progresses).
- **Killing Processes**
  Select a random process in the cycle and kill it! (Would work for jobs such as compiling, but not for a DB system - depends if it is easy to restart the process)

Deadlock Prevention

Attack one of the 4 deadlock conditions.

- **Mutual Exclusion**
  Share the resource

  Issue: This risks race conditions.
- **Hold & Wait** Require all processes to request resources before start, if not available then wait.

  Issue: Need to know what resoruces are needed in advance.
- **No-preemption**
  Issue: Potential issues with race conditions, data corruption etc (e.g forcing a printer to give up half way through a print).
- **Circular Wait**
  Force single resource per process - Issue: Not optimal, requires far more resources.

  Index resources, processes wait for resources in order - Issue: For a large number or resources & processes it is difficult to organise.

Communication Deadlock

Ordering resources & careful scheduling will not help here, instead we must alter the communication protocol to use timeouts.

Livelock

Processes/Threads are not blocked, however the program/system as a whole fails to make progress.

For example a busy wait on a acquiring a mutex, when the mutex cannot be acquired.

```
Mutex A, B;

int main(int argc, char **argv)
{
    // start processes:
    ...
}
```

```
void process_B(void)
{
    enter_region(B);
    enter_region(A);
    // Do stuff
    leave_region(A);
    leave_region(B);
}
```

```
void process_B(void)
{
    enter_region(A);
    enter_region(B);
    // Do stuff
    leave_region(B);
    leave_region(A);
}
```

Or a system receiving messages, may never run a lower priority thread under high load (**receive livelock**).

50004 - Operating Systems - Lecture 9

Oliver Killane

19/11/21

Lecture Recording

Lecture recording is available here

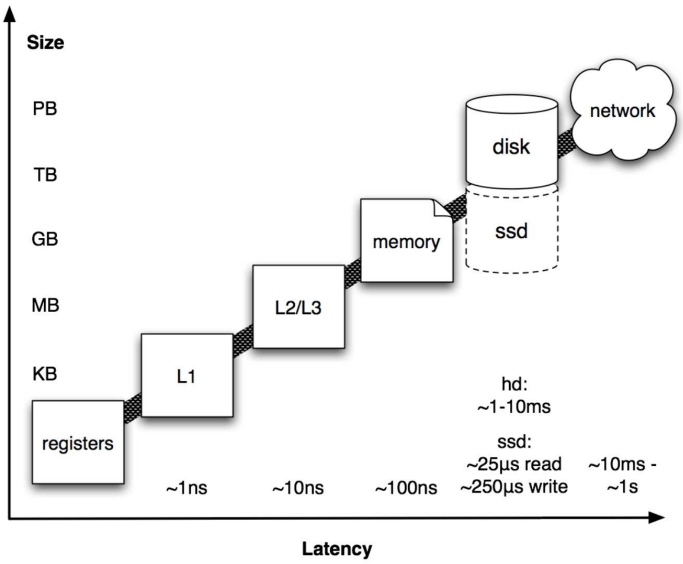
Memory Management

- **Von Neumann Architecture**
  Data and instructions are stored in the same memory (as opposed to Havard Architecture), every instruction requires a memory access.
- **Memory Allocation**
  The OS must be able to allocate memory to programs (i.e malloc/calloc/realloc).

  This needs to be fast & efficient as programs will allocate data on the heap often, and rerquire memory allocated to load.
- **Memory Protection**
  OS must prevent programs for certain memory. This ensures that processes can run in isolation as another process cannot corrupt their memory and cause a crash, likewise with the kernel and use processes.

  This is also important for security as processes may want to hide data (e.g passwords, keys etc) from other untrusted processes.

Memory Hierarchy



Note that the cost (monetary) per size of memory increases from with speed (registers most expensive, disk least).

- **Registers**
  Accessible in a single clock, can store a limited size (usually 64 bits at most) and are very expensive.

  For example **x86_64** architecture uses fully-general purpose 16 registers

- **Cache**
  L3 is typically shared between cores. L2 & L1 per core.

  Caches can have differing levels of associativity, and hold values of recently accessed memory addresses.

Basic Concepts

- Memory Allocation
- Swapping
- Paging (Virtual Memory)
- Page Replacement Algorithms
- Working Set Model

Logical and Physical Addresses

Memory management binds the logical address space to a physical address.

- **Logical Address** Generated by the CPU and the address used by processes.
- **Physical Address** Address used by memory unit, refering to a location in physical system memory.

The use of logical addresses can change based on the **address-binding scheme**.

Address Binding Scheme

Determines when and how logical addresses are bound to physcial addresses, there are 3 such schemes:

- **Compile Time**
  When the program is compiled, the address binding is set.

  The compiler interacts with the OS's memory management.

- **Load Time**
  When a program is loaded into memory, the addresses are bound.

  Binding is done by the OS memory manager (e.g application loader).

- **Execution Time/Dynamic Address Binding**
  Memory binding postponed until execution starts, with binding done by the processor.

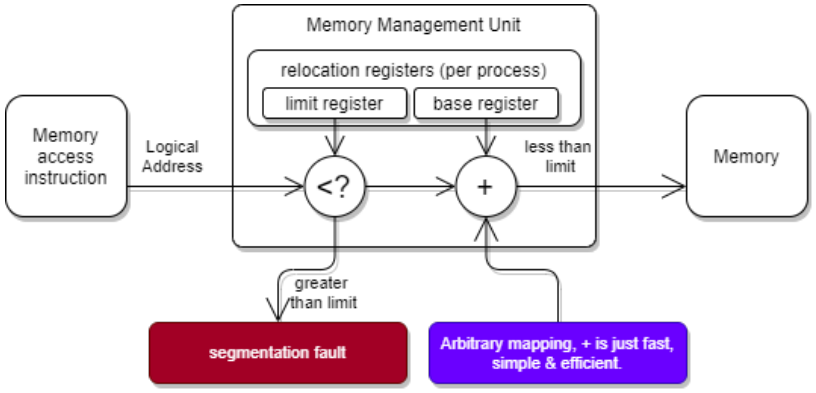
  This is the most common type used by practically all modern operating systems.

In Compile & Load time address-binding, the addresses used by the program are physical addresses. (Program binary is updated to contain these).

In Execution time binding logical addresses are used, and the CPU translates these to physical addresses at runtime.

Memory Management Unit

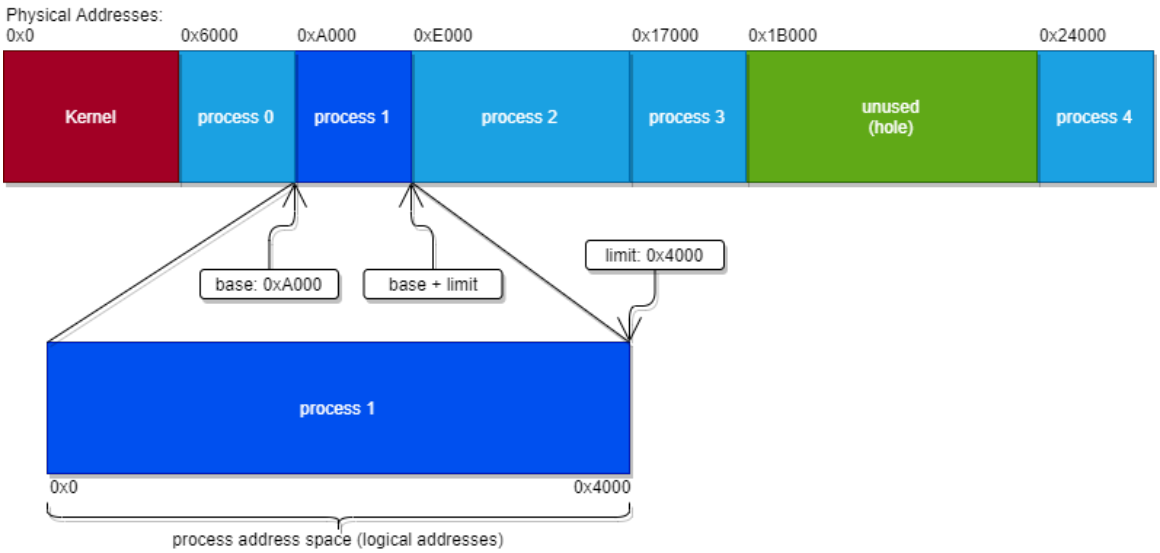
A hardware device to map logical to physical addresses.



The **base register** holds the smallest physical address, the **limit register** holds the highest logical address. This way the **MMU** can ensure:

$$base \leq \text{translated address} < base + limit$$

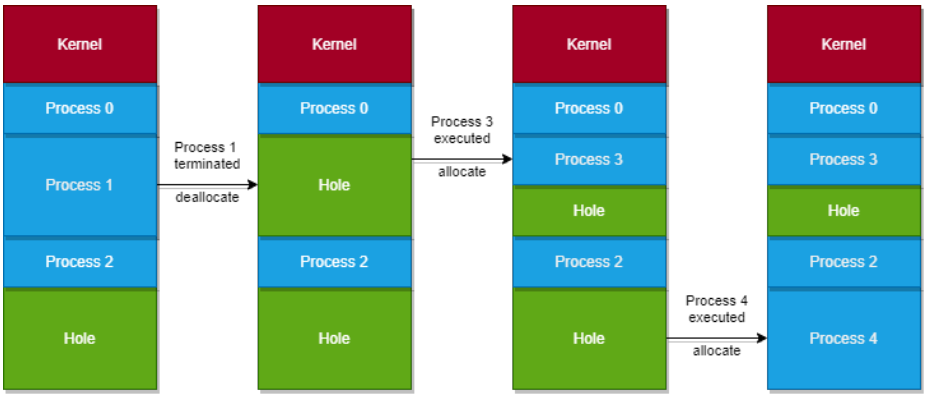
By ensuring processes can only access certain contiguous sections of memory, memory of the kernel and other processes is protected.



Multiple Partition Allocation

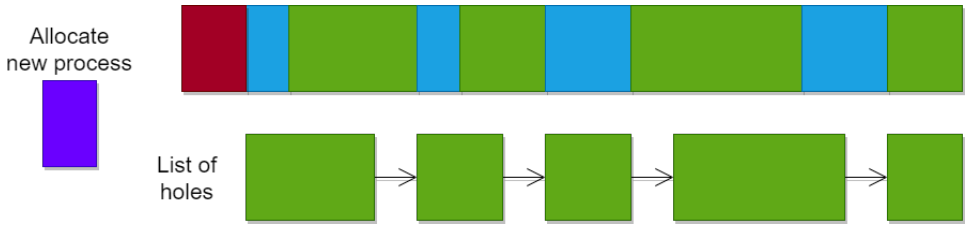
- **Hole** A contiguous section of unallocated memory.
- **Creation/Destruction**
  When a new process is started, the OS allocates the required memory from a sufficiently large hole.

  When a process is terminated, its memory is deallocated, and now free to be allocated to another process.
- **OS Requirements**
  The **OS** must keep track of which partitions have been allocated and which are free (holes).



Dynamic Storage Allocation

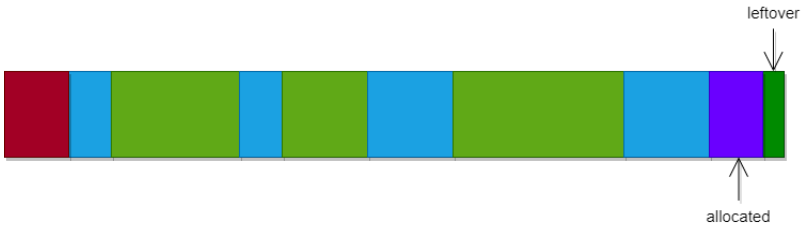
When allocating request for a certain size from the available list of holes and their sizers, held by the OS.



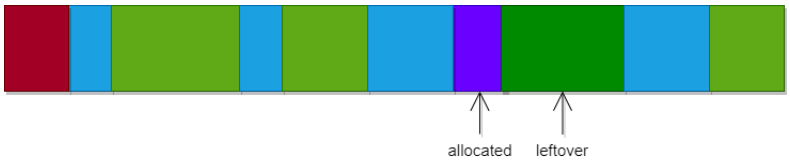
- **First Fit** Allocate in the first hole that is big enough. (Fast & Simple)

- **Best-Fit** Allocate the smallest hole that is large enough.

- Unless the list of holes is ordered, we must traver the whole list to decide which hole to use.
- Produces smallest leftover memory after allocation.



- **Worst-Fit** Allocate largest hole.
  - Search entire list (unless ordered)
  - The largest possible leftover produced.



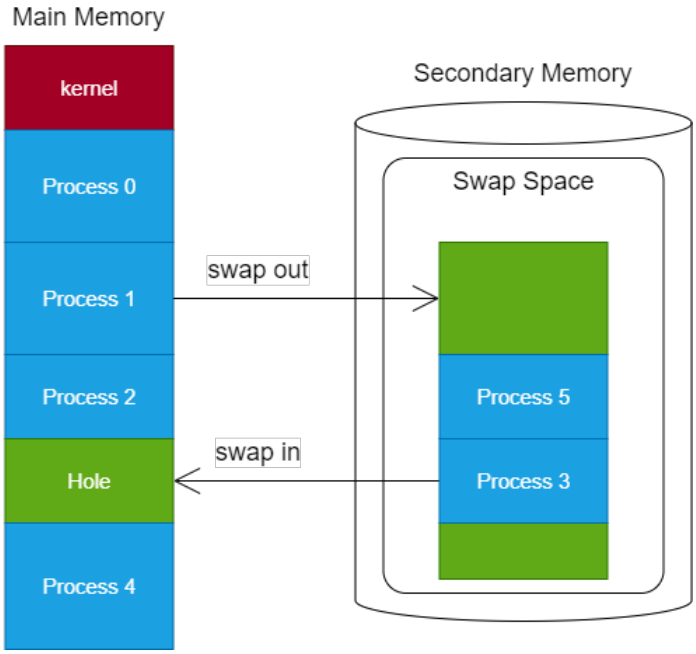
Fragmentation

- **External Fragmentation** Enough memory avaiulable, but not hole large enough. When we have a large number of very small holes. We can fix this by **compaction**:
  - Shuffle memory contents to place all free memory together in one block.
  - Can result in I/O bottlenecks, as requires a very large amount of copying.
- **Internal Fragmentation** Allocated memory larger than the requested. e.g Partition allocated, but program does not make use of all the partition. Wasted memory results in total memory available being low.

Swapping

The number of processes is limited by the amount of available main memory. (Note: only running processes need to be in main memory)

We can supplement main memory with **swap space** (can be a partition or file) on secondary storage (e.g HDD or SSD). However we must be aware than transfer times are long, and we must bring any process that is scheduled back into main memory.

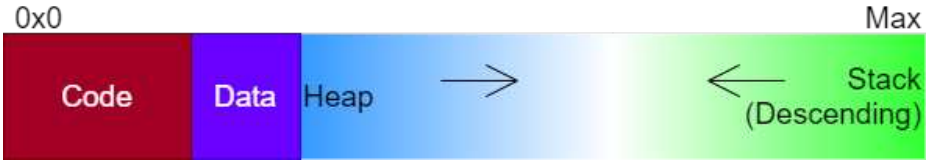


Virtual Memory with Paging

Virtual memory is an abstraction to separate logical memory from physical memory.

- Not all of an executing process needs to be in memory for execution.
- Logical address space can be much larger than the physical address space.
- Address spaces can be shared between several processes.
- Process creation can become more efficient.
- If page size is fixed, external fragmentation is impossible.

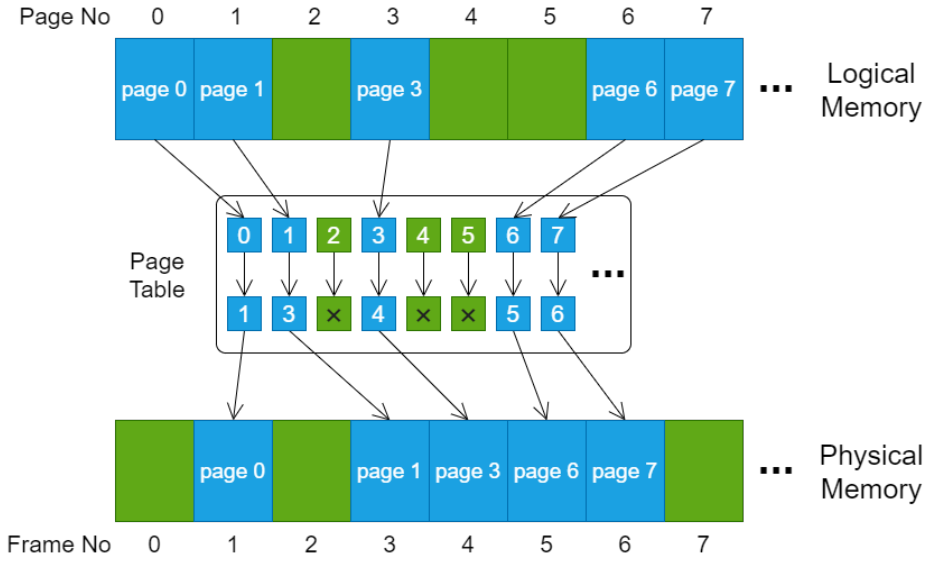
Virtual Address Space



- **Frames**
  Fixed-size blocks of physical memory. The **OS** must keep track of all free frames.
- **Pages**
  A block the same size as a frame, in logical memory (effectively a logical frame).
- **Program Start**
  For a program of size n frames.
  1. Find n free frames, load the program into that memory.

2. Create a page table for the process, mapping logical pages to physical frames.

- **Program Termination** Destroy the page table, freeing all associated frames.



The advantages of this approach are:

- Less internal fragmentation (If a page is not yet used, it is not allocated in memory).
- Less external fragmentation (When allocating alot of memory, split into pages and distribute across physical memory).
- Contiguous logical addresses can be physically non-contiguous.

Page Size

Small Page Size

- Less internal fragmentation.
- Potentially less memory to swap.
- Larger Page table.

Best for efficient memory usage.

Large Page Size

- More internal fragmentation
- Smaller Page table.

Best for low address lookup overhead.

Context Switch

When switching process, the OS must locate the page table of the new process. Set the base register to point to the table in memory, and clear the invalidated cache from the TLB.

TLB

Translation Lookaside Buffer is used to cache the mappings of **Virtual** to **Physical** addresses.

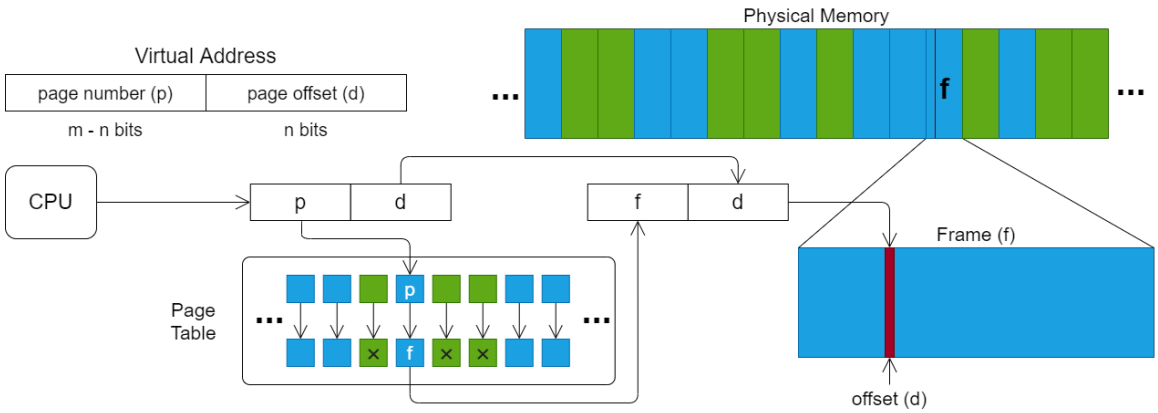
When a virtual address is used & its corresponding physical address found, it is added to the cache so that if the virtual address is used again the physical address does not need to be recomputed.

Address Translation

Each memory address is split into a **page number**(p) and a **page offset**(d)

- **Page Number**
  The index of the page, used to get the base-address of the corresponding frame from the page table.
- **Page Offet**
  The offset through the page/frame, which is combined with the base-address to get the physical address.

For logical address space 2^m with page size 2^n:



For example:
A 16-bit (big endian) system. 1KByte page size. Each page entry's LSB is the valid bit, and second LSB is modified (dorty) bit. Page table:

0	1	2	3	4
$0x2C00$	$0x0403$	$0xCC01$	$0x000$	$0x7C01$

Find the physical memory addresses of the following virtual addresses:

$$0xB85 \quad 0x1420 \quad 0x1000 \quad 0xC9A$$

Get addresses	0xB85	0x1420	0x1000	0xC9A
	↓	↓	↓	↓
Divide address by 1024 to get page #	2	5	4	3
	↓	↓	↓	↓
Get Page Table Entry	0xCC01	PAGE FAULT (past end)	0x7C01	0x000
	↓	↓	↓	↓
Check Valid & Dirty Bits	VALID	N/A	VALID	PAGE FAULT (invalid)
	↓	↓	↓	↓
Divide entry by 1024 to get frame	0x33	N/A	0x1F	
	↓	↓	↓	↓
Add frame onto front of offset	0xCF85	N/A	0x7C00	N/A

Memory Protection

We associate protection bits with each pages.

We associate a valid-invalid bit with each entry in the page table.

- **Valid** Associated page is in process' logical address space.
- **Invalid** Page is missing!
  - Page not in process' logical address space (page fault).
  - Must load page from swap space (on disk) (see demand paging).
  - Incorrect access of some kind.

Demand Paging

Demand paging is a method of memory management where pages in the swap space on the disk are only loaded into main memory when an access attempt is made.

This is opposed to **anticipatory paging** (the norm) where program's pages are loaded from the swap when it starts running, in anticipation of them being accessed.

Page Table Implementation

- **Page Table** is kept in main memory.
- The **Page-Table base register** (**PTBR**) points to the page table's start
- The **page-table length register** (**PTLR**) indicates size.

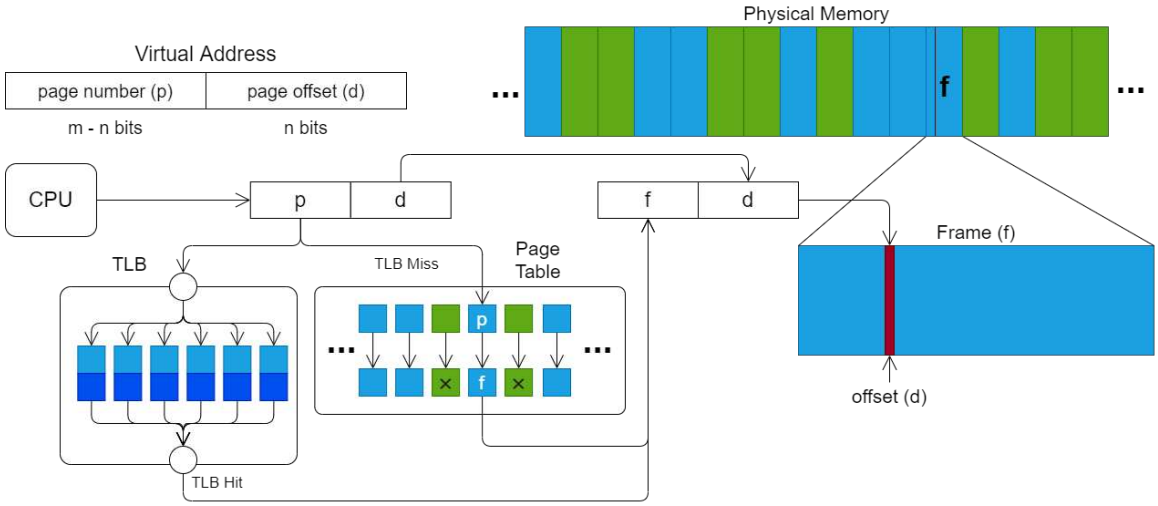
However using the page table for every memory access is slow (requires access to page table in memory), so caching is used.

Note that for kernel mode operations, a bit is set in the page table

Associative Memory

Also called **CAM** (Content addressable memory), it is a special type of memory optimised for search (making use of parallelism).

Generally much more expensive than **RAM** and can only be used for a fixed memory allocation format (cannot search for different data types after creating the silicon).



Translation Look-aside Buffer

An **associative memory** cache that stores translations of virtual pages to physical frames.

Address-Space IDs

Some **TLB**s store **address-space ids** (**ASID**s) in entires. As a result the TLB does not have to be fully wiped when context switching, as it can recognise that old entries are invalid using the associated **ASID**.

This potentially improves performance as entries may survive a short context switch, and no cycles are wasted in removal.

The **MIPS** and **ARM** architecture's TLB uses this approach with 8-Bits ASIDs, and many other architectures have added this feature.

The **reach** of the TLB is the number of addresses that can be cached, for larger page sizes, the reach is larger.

Kernel Page access in System Calls

When a process makes a system call, and the system call handler runs in privileged mode int that process, it may need to access kernel pages.

To ensure safe access, the page table will have a **supervisor** bit. If set for a translation, it means that when in kernel mode, the page can be accessed.

For example with page table:

$Page$	$Frame$	$Valid$	$Supervisor$
0	3	1	0
1	5	1	0
2	14	1	1

A process cannot access page 2 without a segfault, however if in kernel mode following a system call, that page can be accessed.

An alternative implementation would use separate page tables for user space and kernel mode execution.

Effective Access Time

Given that:

- (ω) Memory cycle time.
- (ϵ) Associative Lookup time.
- (α) Hit rate for TLB (also referred to as Associative Registers/Memory) (larger tlb cache $\rightarrow$ higher hit rate)

$$\text{Effective Access Time (EAT)} = (\epsilon + \omega)\alpha + (\epsilon + 2 \times \omega)(1 - \alpha)$$

(If hit, one memory lookup, else 2.)

Page Table Types

Break up the logical address space into multiple page tables:

Multi-Level/Hierarchical Page Table

Page table size can be given by:

$$\text{page table size} = \frac{\text{Address Space Size}}{\text{Page Size}} \times \text{Entry Size}$$

As a result for very large address spaces, with small page sizes, the page table can become large.

This is an issue as:

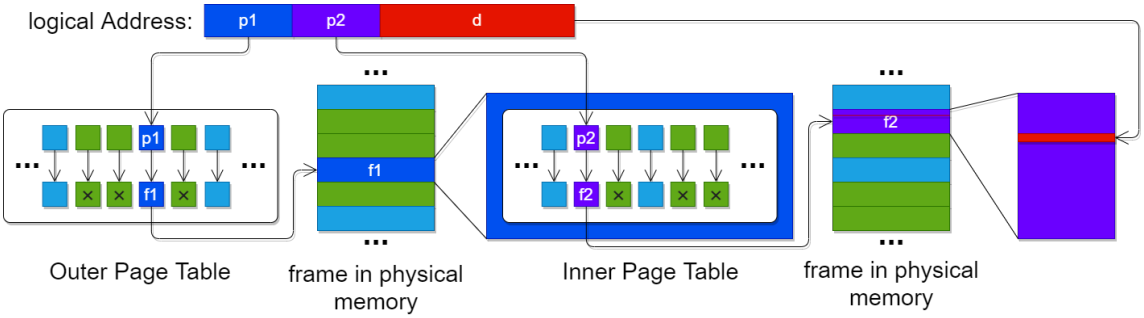
- If page table is larger than a frame - added complexity.
- Page table uses lots of memory, even if a process does not access any/many pages.

To resolve this we can use multi-level page tables - trees containing page tables.

For n levels the page number is partitioned into n sections. To access a page:

1. **Get Outermost Page table** Typically in a single frame.
2. **For each partition of the page number**
   (a) Get index from partition of page number.
   (b) Get entry at index in the current frame (a page table).
   (c) If entry is invalid, empty or dirty, then page fault.
   (d) Else get the frame from the entry. Set this as the current frame.
3. **Get physical address** In the current frame, add the page offset.

An example of a two-level page table is below.



The disadvantage of this system is there are more memory accesses to allow a process to access memory (in a TLB miss).

For example a system with memory access time of $100ns$ and TLB access of $20ns$.

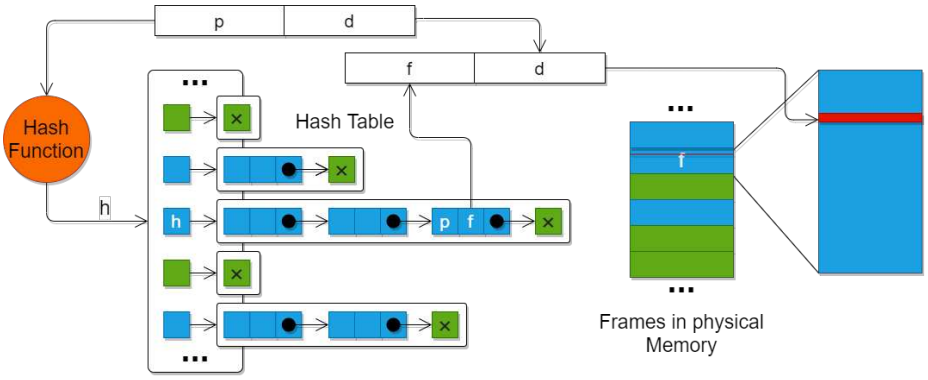
Hit Rate	**Single Level**	**4-Level**
80%	$0.8(20+100)+0.2(20+2\times100)=140ns$	$0.8(20+100)+0.2(20+5\times100)=200ns$
98%	$0.98(20+100)+0.02(20+2\times100)=122ns$	$0.98(20+100)+0.02(20+5\times100)=128ns$

As unpaged memory access requires only $100ns$ we can see the performance of paged memory access gets closer as hit rate increases, even at large paging levels.

Hashed Page Table

A hash table to map page numbers to frame numbers. (A normal page table is effectively a Hashed Page table with identity as the hash function.)

By using more complex hash functions, we can reduce the size of the page table, at the expense of conflicts. Where we can use a linked list to resolve conflicts.



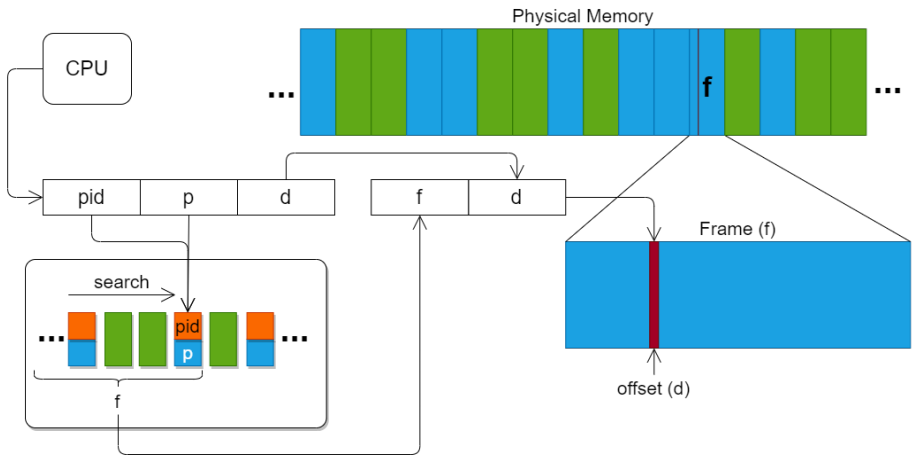
The diagram above uses **bucket based hashmap** using linked lists as buckets. Here entries are hashed and added to an appropriate bucket (with their unhashed key). When hash collisions occur, buckets are filled with more than one entry. When retrieving from hashmap, the hash of the key (for the query) is used to find the correct bucket, and then equality of the keys associated with each entry are used to find the correct entry for the given key.

Inverted Page Table

A page table containing an entry for every page frame of memory, containing:

- Virtual Address of page stored at location.
- Information on the owning process (e.g process ID).

The means only a single page table is used. And it becomes very fast to find the process & page number associated with any given frame (hence the name *inverted*).



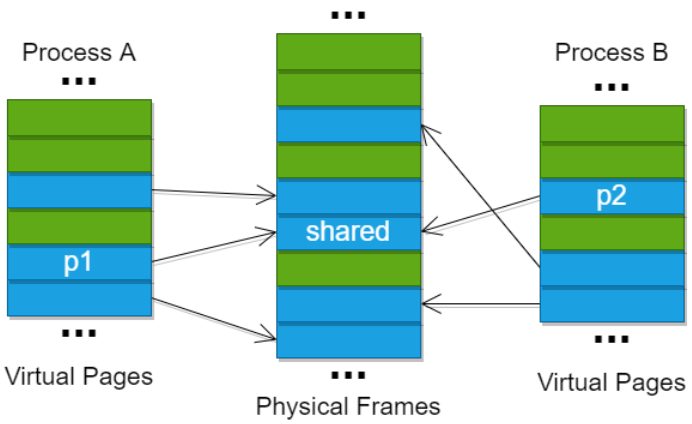
Compared with the standard page table implementation:

- Decreases memory needed to store page table.
- Increased time to search table (e.g must filter/search through enties with other process' IDs).
- Can use a hash table to limit linear search to a few entries.

Shared Memory

Processes can access shared memory by having pages in two processes point to the same frame.

Once the pages are setup to achieve this, the kernel does not need to be involved.



This is useful for IPC, as well as other uses such as sharing libraries.

Comparison with pipes:

- Higher performance (less kernel intervention, and data can be kept in space - less copying)
- Bi-Directional communicqation possible (otherwise would require two pipes).
- Less useful for unidirectional communication as kernel provides synchronisation for pipes.

For example **System V**'s API (note more modern systems use **mmap**):

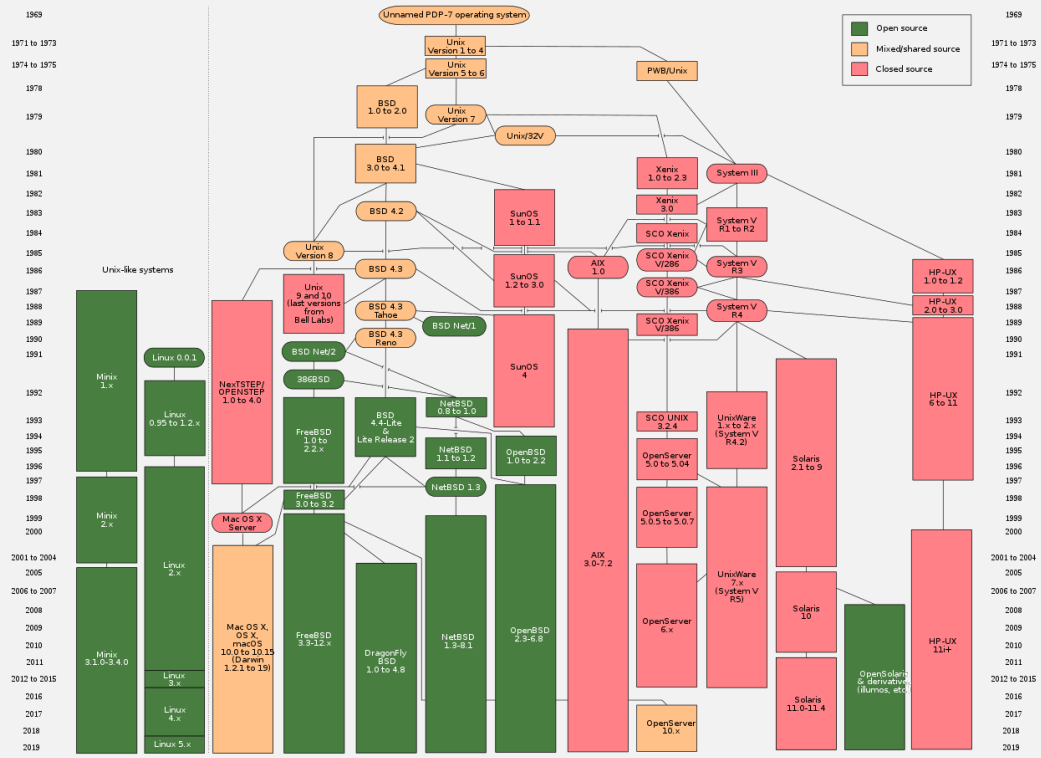
```
1  #include <sys/ipc.h>
2  #include <sys/types.h>
3  #include <sys/shm.h>
4
5  /* To use the System V IPC, you need a key from an associated file */
6  key_t ftok (const char *pathname, int proj_id);
7
8  /* Get a memory  sahred memory segment with key, or create a new one.
9   * There are several flags for creating private shared memory, and setting
10  * permissions (use shmflag).
11  */
12 int shmget (key_t key, size_t size, int shmflg);
13
14 /* Attach shared memory segment to the address space of a process. */
15 void *shmat (int shmid, const void *shmaddr, int shmflg);
16
17 /* Detach a share memory segment from the process. */
18 int shmdt (const void *shmaddr);
19
20 /* Issue a command to control the shared memory segment. */
21 int shmctl (int shmid, int cmd, struct shmid_ds *buf);
22
23 struct shmid_ds {
24     struct ipc_perm shm_perm;     /* Ownership and permissions */
```

```
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Last change time */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};

struct ipc_perm {
    key_t          __key;    /* Key supplied to shmget(2) */
    uid_t          uid;      /* Effective UID of owner */
    gid_t          gid;      /* Effective GID of owner */
    uid_t          cuid;     /* Effective UID of creator */
    gid_t          cgid;     /* Effective GID of creator */
    unsigned short mode;     /* Permissions + SHM_DEST and
                                SHM_LOCKED flags */
    unsigned short __seq;    /* Sequence number */
};
```

System V

One of the first commercial **Unix** Operating systems originally developed by **AT&T** and released in 1983.

Was a competitor/rival to **BSD** with much cross pollination fo features occurring between the two. It also had several distributions by other companies such as Oracle's Solaris OS.



50004 - Operating Systems - Lecture 10

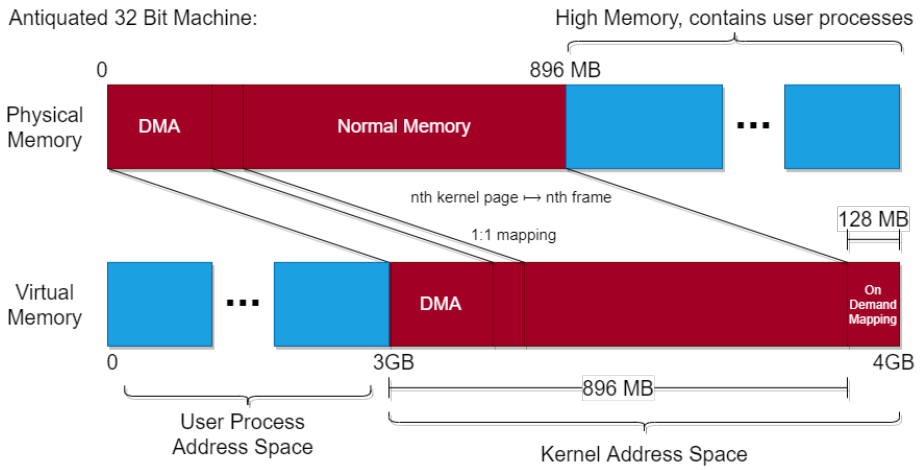
Oliver Killane

22/11/21

Lecture Recording

Lecture recording is available here

Linux - Virtual Memory Layout



- 1 : 1 **Mapping**
  Can turn logical address to physical address for kernel pages by subtracting 3GB.
  - Very efficient for kernel memory access
  - Does not require change of page table (not changing context such as in process switching) (so TLB is not flushed when user process makes syscall).
  - On Demand Mapping contains temporary mappings for use of more than 896 MB of memory.

Typically on IA32

- 4KB page size.
- 4GB virtual address space.
- Two-level page table (up to 3 with **Physical Address Extension**).
- Offset bits contain page status (dirty, read-only etc).

On AMD64/x86_64

- Larger page sizes (e.g 4MB).
- Up to four-level page table.
- Offset bits can contain can-execute (prevent malicious code being written to take over process).

Meltdown Attack

```
1  s = a + b
2  t = s + c
3  u = t + d
4  v = u + e
5  if (v == 0) { /* Context speculativeley executed. */
6      w = kernel_mem[addr]
7      x = w & 0x1;
8      y = x * 4096;
```

```
    z = user_mem[y];
}
```

By speculative execution, this becomes:

```
/* ensure user_meme[0...4096] is not in cache*/

s = a + b
t = s + c
u = t + d
v = u + e

/* speculative execution*/
w_ = kernel_mem[addr]  /* no page fault for speculative */
x_ = w & 0x1;
y_ = x * 4096;
z_ = user_mem[y];

/* If statement true, speculative results are set */
if (v == 0) {
    w = w_;
    x = x_;
    y = y_;
    z = z_;
}

/* check how long it takes to read user_mem[0] and user_mem[4096] to determine
 * if it is in cache.
 *
 * If mem[0] has been cached, then kernel_mem[addr]'s LSB is 0
 * If mem[4096] has been cached, then kernel_mem[addr]'s LSB is 1
 */
```

Demand Paging

Only load pages from swap when the user attempts to access them.

- Lower I/O load (unused pages are never loaded)
- Less memory required (fewer pages resident in memory)
- Faster response time (Can start executing straight away, does not need to wait for all pages to be loaded first)
- Supports more users (lower memory usage allows this)

Uses a valid-invalid bit such that:

$$\begin{aligned} 1 \Rightarrow &\quad \text{in memory} \\ 0 \Rightarrow &\quad \text{not in memory} \end{aligned}$$

- All page entries are originally set to 0.
- If a page with bit 0 is accessed, page fault and trap to kernel.
- Kernel uses other table to determine if reference is invalid, or valid but page not in memory.

To handle a valid request the kernel:

1. Get empty frame
2. swap page to frame
3. Reset tables (validation bit = 1)
4. Restart last instruction

Performance

Given a **Page Fault Rate** ($= 0 \Rightarrow Never$, $= 1 \Rightarrow Always$).

$$\text{Effective Access Time} = (1-p) \times \text{memory access time} + p \times \left(\begin{array}{c} \text{Page Fault overhead} \\ + \\ \text{swap page out} \\ + \\ \text{swap page in} \\ + \\ \text{restart overhead} \\ + \\ \text{memory access time} \end{array} \right)$$

Virtual Memory Tricks

- **Copy-on-Write (COW)**
  Processes accessing identical pages use the same frame, only copy when a process wants to write/modify the page.

  – Parent and Child processes can initially share same pages in memory
  – Efficient process creation (copy only modified pages)
  – Free pages allopcated from a pool of zero-ed out pages

  For example with **fork**:

  – Child's page table points to parent's pages (marked as read-only in child & parent's page table)
  – Protection fault causes trap by kernel.
  – Kernel allocated new copy of the page to process alterning (that it can write to), replaces old page in page table.
  – Both child and parent's page table sets page to Read-Write.

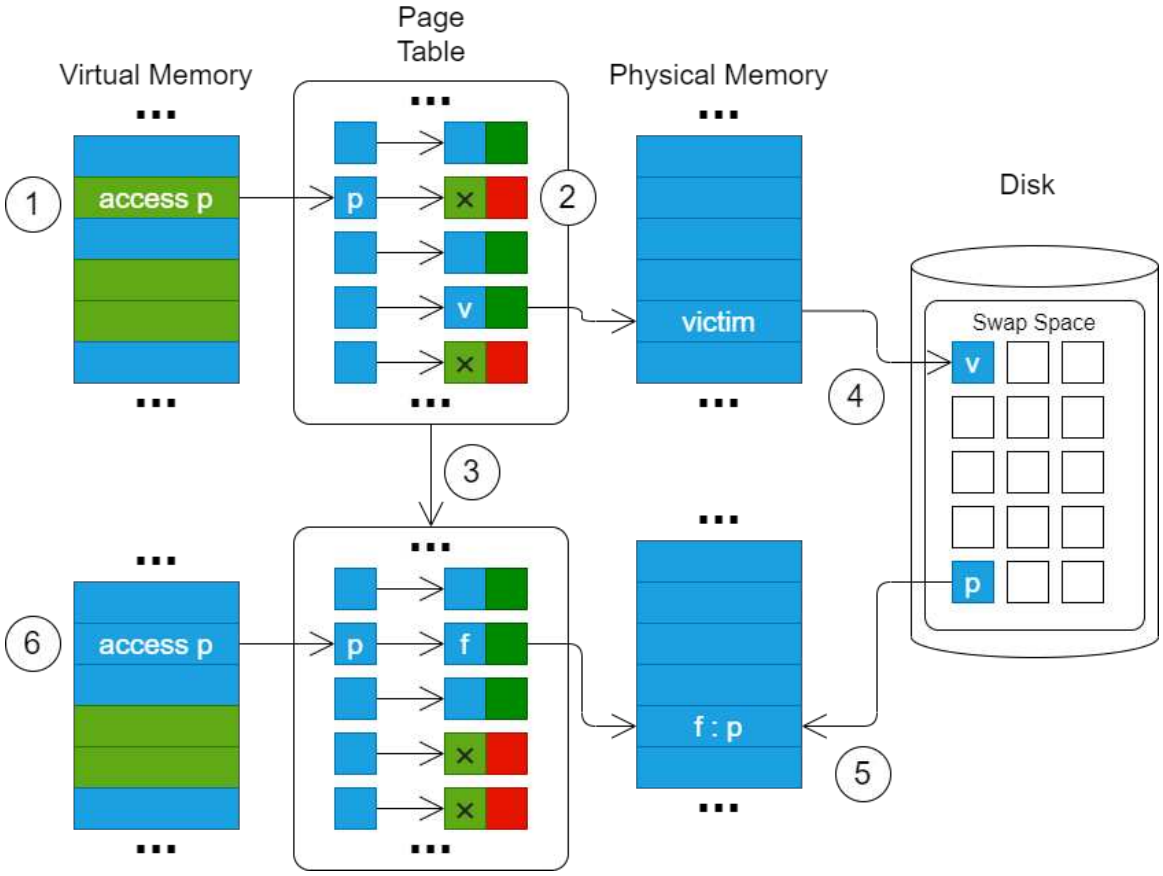
- **Memory Mapped Files**
  Map files into the virtual address space using paging. Only need to load parts of a file when they are accessed.

  e.g 16K page file, only access first 2 pages, so only first 2 loaded to memory.

  Simplified programming model for I/O (can easily access stdin/out).

Page Replacement

When out of free memory & a new page must be created, a page must be swapped out.
An example below (note victim does not have to be a page of the same process).



1. Access a page not loaded into memory.
2. Page table contains bit to show page is not loaded.
3. Update page table (must ensure no race conditions when involving multiple processes that may be running on multiple cores).
4. Write victim to disk (swap space).
5. Read page from disk.
6. Restart operation, can now access page.

The goal is to:

- **Reduce the number of page faults**
  Avoid bringing a page back into memory many times & in general more frames ⇒ fewer page faults.
- **Prevent over-allocation of memory**
  Page-fault service routine should include page replacement.
- **Reduce redundant I/O**
  Use modify (dirty bit) to only load modified pages back to disk.

Replacement Algorithms

FIFO

Replace the oldest page.

Advantages

- Simple to implement

Disadvantages

- May replace a heavily used page.

Belady's Anomaly (where more frames result in more page faults), for example:

3 frames, 9 faults

Access:	1	2	3	4	1	2	5	1	2	3	4	5
Page Fault:	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N
Frame 0:	1	1	1	4	4	4	5	→	→	5	5	→
Frame 1:	N	2	2	2	1	1	1	→	→	3	3	→
Frame 2:	N	N	3	3	3	2	2	→	→	2	4	→

4 frames, 10 faults

Access:	1	2	3	4	1	2	5	1	2	3	4	5
Page Fault:	Y	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y
Frame 0:	1	1	1	1	→	→	5	5	5	5	4	4
Frame 1:	N	2	2	2	→	→	2	1	1	1	1	5
Frame 2:	N	N	3	3	→	→	3	3	2	2	2	2
Frame 3:	N	N	N	4	→	→	4	4	4	3	3	3

Here the reference string is such that under **FIFO** at 4 frames the next page is frequently the oldest, resulting in a high number of page faults.

Optimal Algorithm

Replace the page that will not be used for the longest period of time. This is impossible in practice, but can be used as a benchmark for comparing other algorithms.

Note that in the example, pages never used again are the longest away & lower frames are chosen when pages have equal time.

3 frames, 7 page faults

Access:	1	2	3	4	1	2	5	1	2	3	4	5
Page Fault:	Y	Y	Y	Y	N	N	Y	N	N	Y	Y	N
Frame 0:	1	1	1	1	→	→	1	→	→	3	4	→
Frame 1:	N	2	2	2	→	→	2	→	→	2	2	→
Frame 2:	N	N	3	4	→	→	5	→	→	5	5	→

50004 - Operating Systems - Lecture 11

Oliver Killane

23/11/21

Lecture Recording

Lecture recording is available here

Page Replacement Algorithms Continued...

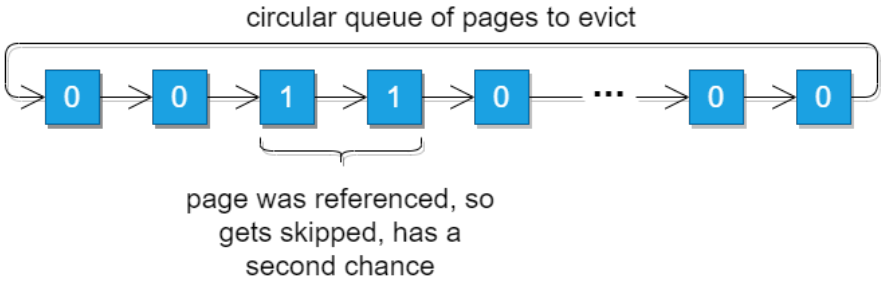
Least Recently Reused Page

Each page has a counter, when referenced, copy clock to counter. When replacing, choose page with lowest counter.
3 frames, 9 page faults

Access:	1	2	3	4	1	2	5	1	2	3	4	5
Page Fault:	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N
Frame 0:	1	1	1	4	4	4	5	→	→	5	5	→
Frame 1:	N	2	2	2	1	1	1	→	→	3	3	→
Frame 2:	N	N	3	3	3	2	2	→	→	2	4	→

As proper **LRU** is expensive (requires a search for lowest counter & to store counters), so we use approximations.

- **Reference Bit**
  - Each page has reference bit r, inbitially $r = 0$.
  - When a page is referenced $r = 1$ (done by **MMU**).
  - Periodically reset bits.
  - When evicting, choose a page with $r = 0$.
- **Second Chance** Evict pages in order, but if $r = 1$ give a second chance to stay in memory.



  - Uses a reference bit r and usesd clock replacement
  - If page to be replaced (in clock order) has $r = 1$, set $r = 0$, leave in memory, search for another page.

Counting algorithms

Work by keeping a counter of the number of references.

- **LFU - Least Frequently Used**
  - Replace page with the smallest count.

  - May replace very new page just brought into memory.
  - Never forgets heavy page usage (reset counters or use ageing)

- **MFU - Most Frequently Used**
  - Replace page with largest count.
  - Newly accessed pages have low count, are prioritised.
  - Pages barely used hog memory & heavily used pages evicted quickly.

Locality of Reference

Programs tend to request the same pages across space & time. We need to design for this to ensure good performance.

If we do optimise for this, it could result in **thrashing**

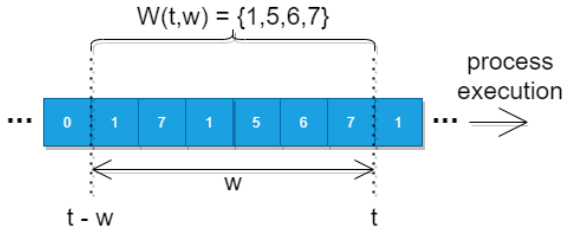
Thrashing

where memory virtual memory is so overused, that there are constant page faults, and paging routines (OS replacing). This destroys performnance.

- Excessive page faults & page replacement causing low CPU utilisation, low performance.
- Program gets slowed by constant access to slow secondary storage.

Working Set Model

A **working set** of pages $W(t, w)$ is a set of pages referenced by a process while running at time interval $t - w$ to t.



We can then use this in our clock replacement algorithm, keeping track of pages in a working set.

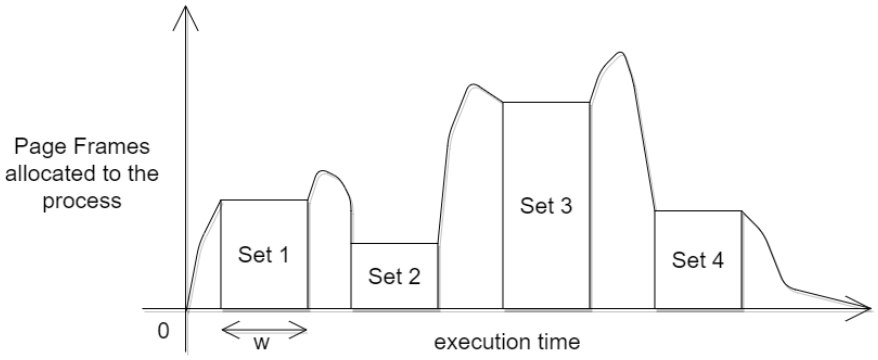
At each page fault:

1. $r = 1$ Set $r = 0$ and move to next page.
2. $r = 0$
   (a) Calculate age.
   (b) If $age < w$ (working set age) continue (page is in working set).
   (c) If $age > w$, if page clean, replace, else start write back and continue to find another page (once the write-back of the frame is complete the page is marked as free & clean - we dont want to halt while waiting for the page to be written back to disk)

Effectively we only replace pages if they haven't be referenced in w time, so that we can take advantage of temporal locality.

Working Set Size (w)

Processes transition between working sets (different parts of the program use different sets of pages). Hence allocations tend to look like:



Here we do not deallocate pages referenced within w time, at different points in the program the number of allocated page frames differ.

In transitions the number of allocated pages tend to be higher as pages from two sets are referenced within w time.

- Working set too large - too many allocations, process uses too much memory.
- Working set too small - tooo few allocations, lots of page faults, slow.
- Programs transition between working sets (e.g program moves to a new phase, uses different structures).
- Dont want to misallocate - OS allocates for pages the program dosen't actually want anymore.

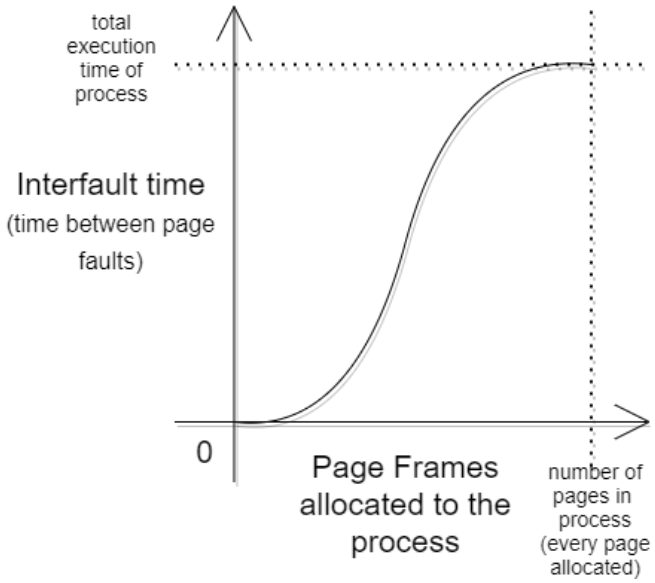
A program can transition between working sets, e.g:

```
1  int main(int argc, char **argv)
2  {
3      /* assume struct big uses memory over several pages. */
4      struct big big1;
5      struct big big2;
6
7      init_big_structure(&big1);
8
9      /* first working set. Using pages associated with big1. */
10     while(some_condition) {
11         /* code using the big1. */
12     }
13
14     /* transition, e.g copying from big1 to big 2, using pages from both. */
15
16     /* second working set. Using pages associated with big2.*/
17     while(some_condition) {
18         /* code using the big2. */
19     }
20 }
```

As we allocate more pages, the number of page faults will decrease (so interfault time will increase). When we allocate all pages in the process there will be no page faults, so interfault time is execution

time.

This generally follows:



Local and Global Page Replacement

- **Local Strategy**
  When replacing a page, a process can choose some page that belongs to the same process.

  Requires some partitioning to determine how many pages each process owns/can be allocated. The most popular techniques are:

  – fixed partitioning - each process gets a fixed size
  – balanced set algorithms

  As processes manage page faults independently, it is more scalable than global (at most considering one process' pages when replacing).

- **Global Strategy**
  Can choose any page from any process.

  – memory is dynamically shared between processes.
  – Initially allocate memory proportional to process size.
  – Use page fault frequency to tune allocation (more page faults → more allocation).

  Uses all pages to determine replacement (slower).

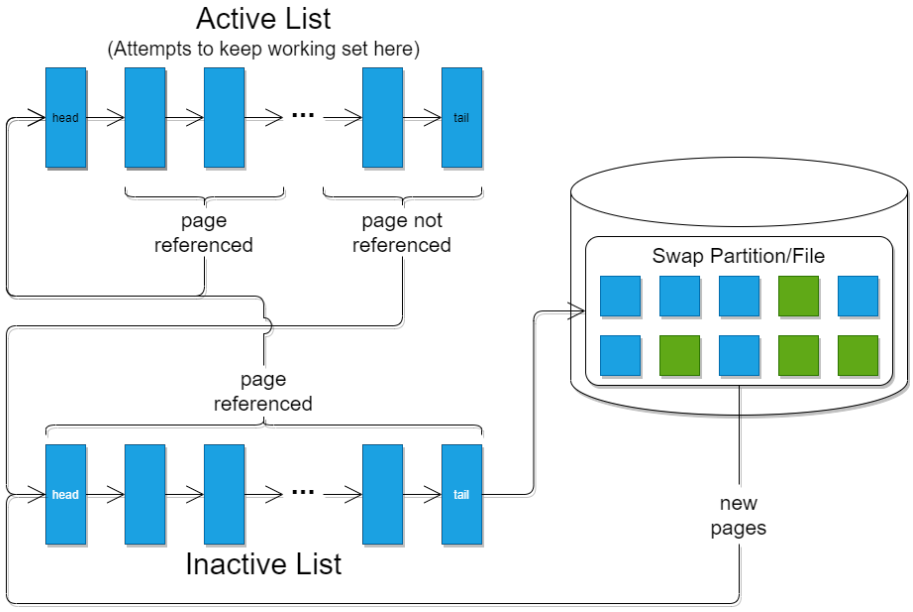
  More efficient overall than local. e.g demanding process's page replaces page of process idling/not using the page.

There is no agreed best solution & it is partially dependent on the scheduling strategy.

- Windows is local
- Linux is global

Linux Page Replacement

Uses a variation of the **clock algorithm** to approximate **LRU** page replacement.



- **kswapd** Swap Daemon
  - Pages in inactive list reclaimed when memory low.
  - Uses dedicated swap partition or file
  - Handles locked & shared pages.
- **pdflush** Kernel Thread
  - Periodically flushes dirty pages to disk from the inactive list.
  - Clean pages can just be freed, no need to write back.
- **Reference bit** Used to determine when pages are moved between active & inactive lists.

Example:

Given 3 empty frames and a reference string of virtual memory accesses:

1 2 1 3 2 1 4 3 1 1 2 4 1 5 6 2 1

Assume demand paging. **LRU replacement policy - 11 page faults**

Access:	1	2	1	3	2	1	4	3	1	1	2	4	1	5	6	2	1
Page Fault:	Y	Y	N	Y	N	N	Y	Y	N	N	Y	Y	N	Y	Y	Y	Y
Frame 0:	1	1	→	1	→	→	1	1	→	→	1	1	→	1	1	2	2
Frame 1:	N	2	→	2	→	→	2	3	→	→	3	4	→	4	6	6	6
Frame 2:	N	N	→	3	→	→	4	4	→	→	2	2	→	5	5	5	1

Second chance with clock algorithm - 9 page faults

Access:		1	2	1	3	2	1	4	3	1	1	2	4	1	5	6	2	1
Page Fault:		Y	Y	N	Y	N	N	Y	N	Y	N	Y	N	N	Y	Y	N	Y
Frame 0:	N	$1_{r=1}$	$1_{r=1}$	→	$1_{r=1}$	→	→	$4_{r=1}$	→	$4_{r=1}$	→	$4_{r=0}$	$4_{r=1}$	→	$5_{r=1}$	$5_{r=1}$	$5_{r=1}$	$5_{r=0}$
Frame 1:	N	N	$2_{r=1}$	→	$2_{r=1}$	→	→	$2_{r=0}$	→	$1_{r=1}$	→	$1_{r=0}$	→	$1_{r=1}$	$1_{r=0}$	$6_{r=1}$	$6_{r=1}$	$6_{r=0}$
Frame 2:	N	N	N	→	$3_{r=1}$	→	→	$3_{r=0}$	$3_{r=1}$	$3_{r=1}$	→	$2_{r=1}$	→	→	$2_{r=0}$	$2_{r=0}$	$2_{r=1}$	$1_{r=1}$

50004 - Operating Systems - Lecture 12

Oliver Killane

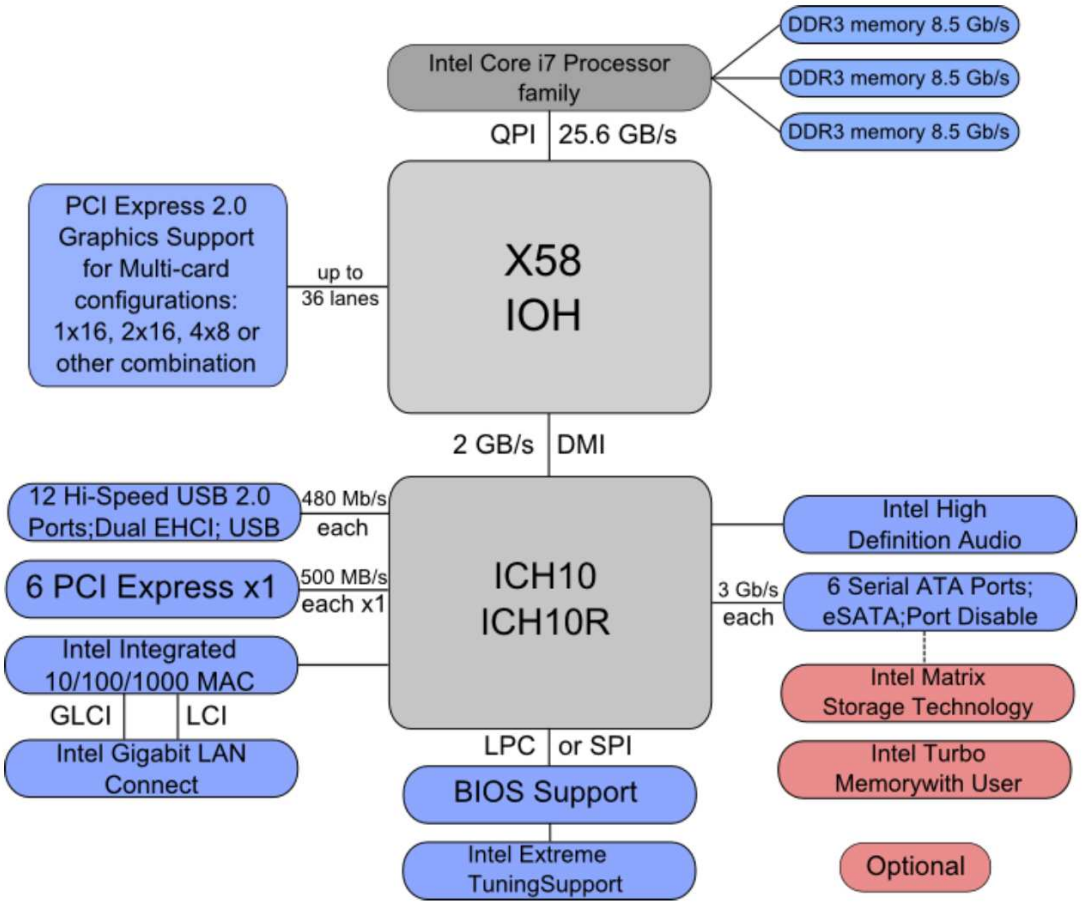
25/11/21

Lecture Recording

Lecture recording is available here

Device Management

Intel Example



North & South Bridge

Here the **X58** connects the CPU to high-speed peripherals through the high bandwidth **QuickPath Interconnect** (**QPI**). This is the **North Bridge** which connects the CPU to peripherals directly.

The **ICH10** (I/O Controller Hub) supports lower speed peripherals, and connects to the CPU through the **North Bridge**.

I/O Device Management Objectives

- **Fair Access to Shared Devicess** Prevent processes hogging resources
- **Exploit Parallelism**
  Can use devices in parallel (e.g send packets using network card while writing to disk), and some devices have parallelism themselves.
- **Provide uniform & Simple view of I/O**
  Abstract devices away from processes (e.g filesystem, not disk). And use uniform interfaces (when new devices added, programs do not need to change to utilise).
  - Uniform naming and error handling.
  - Hide complexity of device handling.

Device Independence

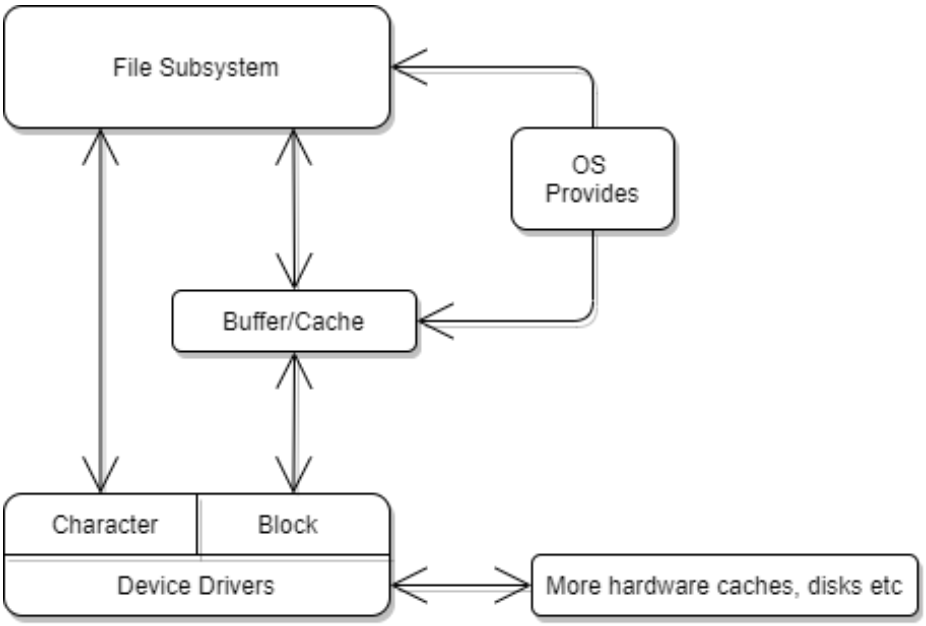
Have the device be independent from its type (e.g terminal, disk, dvd drive) and which instance (e.g disk 1, 2, 3).

E.g can use the same interface for all disks connected to a system. Can read data from dvd drive, disk, terminal in a single interface (sometimes we split into classes when differences are large enough).

Device Variations

- **Unit of Data Transfer** Character (bytes) or block
- **Supported Operations** e.g read, write, seek
- **Synchronous or asynchronous** e.g network card (send request, then get result back sometime), disk (read and block until the data is read)
- **Speed differences** e.g NVMe SSD vs Tape Drive
- **Shareable** e.g disks are shareable, printers are not (can print one at a time)
- **Types of error conditions** e.g Disk errors, vs GPU temperature warnings

Character vs Block Device



Note the type (character/block) depends on the type of device.

Block	Maximize throughput	Disks, network cards etc
Character	Minimize latency	Keyboard, terminal etc

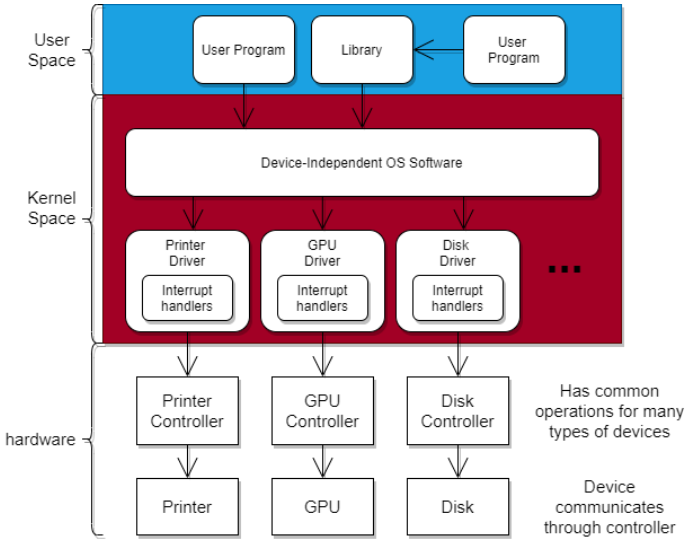
Character Devices:

1	mem	Device file representing the physical memory of the os.
2	pty	Pesudoterminal (bidirectional communication channel).
180	usb	USB devices.

Block Devices:

1	ramdisk	access ram disk in raw mode.
2	fd	Floppy Disk.
7	loop	Loop device, maps data blocks to a file in the filesystem, or another block device.

I/O Layers



Interrupt Handler

An interrupt is a signal sent from a device to the CPU to inform it that the device needs attending to (e.g device connecting, finished reading, error has occured etc.).

Drivers register handlers to deal with types of interrupts, when the CPU receives an interrupt, it runs the relevant handler.

- For block devices, on transfer completion signal device handler.
- For character devices, when character transfered, process next character.

For modern systems (e.g nvme storage) a hybrid approach makes use of the following:

- Polling (queue of requests and responses), very fast but uses CPU time (analogous to spin locks)
- Wait for interrupt (better for longer waits - another process can be scheduled)

Device Driver

Handles a type of device, can control multiple devices of the same type.

- Implements block read/write
- Access device registers (writing control information to a device)
- Initiating Operations (start device e.g at boot)
- Scheduling requests (if a device is shared, placing in order, often for performance - order hard disk operations to do the least disk traversal).
- Handle Errors

Device Independent OS Layer

Use standard interfaces for drivers of device types:

- Simplifies OS design.
- Interface to write new drivers to.
- No OS changes required to support new drivers, just support interfaces.

This layer also provides **device independence**:

- **Map Logical to Physical Devices** (Naming and Switching)
  Can map one logical device to many physical, or vice versa (e.g disk RAIDs).
- **Request Validation against device**
  Check device & driver is working correctly.
- **Allocation**
  Determine which processes can access which logical devices.

  Control **dedicated allocation** (process exclusive access to a device)
- **Buffering**
  As previously mentioned - for performance & block size independence.
- **Error Reporting**

User-Level I/O Interface

System call interface to allow user programs to interact (often through other 3rd party libraries).

- basic I/O operations (**close**, **read**, **write**, **seek**)
- Sets up parameters (device independent)
- Can be synchronous (blocking) or asynchronous (non-blocking)

Unix files

Unix accesses virtual devices as files. This allows access to devices using the normal standard input/output calls. For example the common:

File Descriptor	Name
0	Standard Input
1	Standard Output
2	Standard Error

There are also files such as **/dev/kdb** for keyboard access.

Device Allocation

- **Dedicated Device** (e.g DVD writer, terminal, printer)
  - One process gets exclusive access to a device.
  - Typically allocated for long periods of time (e.g minutes - hours: printing, or controlling a terminal display output)
  - If another process tries to access, it fails (potentially adding to a queue of open requests).
  - Only allocated to authorised processes (e.g don't want malicious processes blocking access to a resource).

- **Shared Device** (e.g disks, window terminals etc)
  OS can provide systems for sharing, e.g file system accessible by all processes makes use of the disk.
- **Spooled Device** (e.g printer, dvd writer - many other dedicated devices)
  A shared pool. A **daemon** process has dedicated access, processes send requests/jobs to the **daemon**.
  - Provides sharing on non-sharable resources.
  - Reduces I/O time resulting in greater throughput.

Buffered vs Unbuffered I/O

- **Buffered**

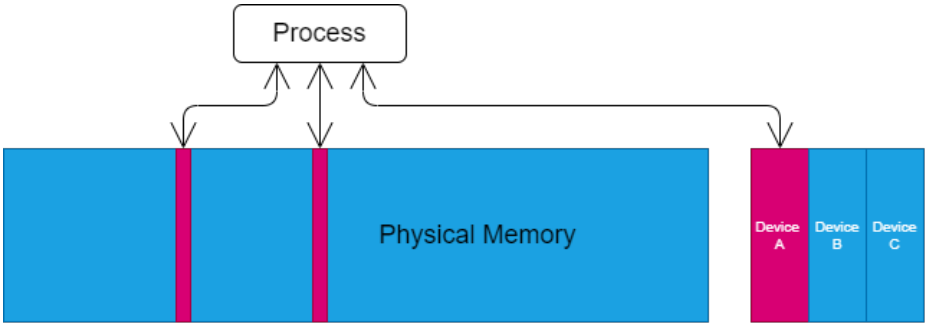
Output	User data transferred to OS output buffer. Process continues and only suspends once buffer is full.
Input	OS reads ahead. Reads taken from buffer, process blocks when buffer empty.

  - Smooths peaks in I/O traffic (allows for limited load balancing).
  - Can allow for different data transfer unit sizes between devices (e.g buffer contains blocks).
- **Unbuffered**
  - Data Transfered directly between device and user space.
  - Each read/write causes physical I/O (device does something, not just accessing a hidden buffer).
  - Device handler used for every transfer.
  - High switching overhead (e.g every read requires the driver to take over, and to do some physical action).

Device Drivers

Memory Mapped I/O

Device can be addressed as a special memory location.



Hence we can use the virtual memory setup to restrict access (e.g set supervisor bit to prevent user access).

I/O

- **Programmed I/O** (simple but inefficient)
  Wait for device (spin), then continue execution.
- **Interrupt Driven** (large overhead, good for long expected waits)
  Hardware sends an interrupt when operation completed, can do other work while waiting.
- **Direct Memory Access (DMA)** (requires hardware, but reduces CPU intervention)
  A DMA controller (often in the device) waits for the device to respond, then once the full result is available, places this directly into memory.

Linux

Loadable Kernel Module (LKM)

Device drivers are loadable modules that are loadeed and linked dynamically with the running kernel.

This requires binary compatability (module must be specific to kernel version).

Linux uses **Kmod**

- Kernel subsystem that manages modules without user intervention.
- Determines modules dependencies.
- Loads modules on demand (e.g load driver for network card when it is connected to the system).

Basic LKM module

```
1  /* Used for all initialisation code. */
2  int init_module (void)
3  {
4      ...
5  }
6
7  /* Used for clean shutdown */
8  void cleanup_module (void)
9  {
10     ...
11 }
```

Kernel can open file, look for symbol table (generated by compiler), and call the corresponding functions.

```
1
2  # insmod loads a module to the kernel
3  # 'sudo' as this operations is restricted to the root user (access to all
4  # commands and files).
5  sudo insmod some_module.o
```

IO Management

/dev /sys /proc

Linux (in UNIX fashion) uses files to represent many drivers, services & methods to collect system information. Some of the main directories for this are:

- **/dev** device file directory
  Contains files for devices (virtual included).
- **/proc** virtual filesystem for processes
  Contains information on running processes, each represented by files of size 0. Each numbered directory is actually the **pid** of a running process.

  Other files can be read to get system information such as:

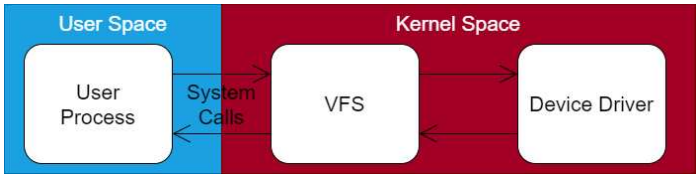
/proc/meminfo	Information on the memory system including free pages, kernel stack pages etc.
/proc/interrupts	Number of interrupts received for categories.
/proc/stat	System status information (e.g number of processes).
/proc/version	OS version information.

- **/sys**
  A filesystem like view of the kernel and its configuration/settings.

- **Device Classes** Group similar types of devices (similar function, performance needs)
- **Identification Numbering** $< Major >:< Minor >$
  Major Determines which diver is controlling. (e.g IDE Disk Drive, Serial port etc.)
  Minor Distinguishes devices of the same class. (e.g Drive Number)
  Examples from kernel documentation are here.
- **Special Files**
  Most devices are represented by device/special files in the **/dev** directory.

Device Access

Device files are accessed via the **virtual file system (VFS)**.



Most drivers implement **read**, **write**, **seek** etc (much like a file), however all contain other operations which do not fit this abstraction.

Linux uses the **ioctl** (I/O Control) system call to support special tasks (e.g getting printer status, ejecting a CD drive)

```c
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);

/* for example to eject a CD-ROM */
ioctl(cddrom, CDROMEJECT, 0);
```

Character Device I/O

- Data transitted as a stream of bytes (read a byte, then another is presented)
- Represented by a **char_device_struct** structure.
- **char_device_struct** contains a pointer to a **file_operations** struct.

The **file_operations** struct:

- Maintains operations supported by the device driver.
- Stores function pointers to operations (read, write etc).

file_operations in the linux kernel (github).

```
struct file_operations {
        struct module *owner;

        loff_t    (*llseek)         (struct file *, loff_t , int);
        ssize_t   (*read)           (struct file *, char __user *, size_t , loff_t *);
        ssize_t   (*write)          (struct file *, const char __user *, size_t ,
            ↪ loff_t *);
        ssize_t   (*read_iter)      (struct kiocb *, struct iov_iter *);
        ssize_t   (*write_iter)     (struct kiocb *, struct iov_iter *);
        int       (*iopoll)         (struct kiocb *kiocb, struct io_comp_batch *,
            ↪ unsigned int flags);
        int       (*iterate)        (struct file *, struct dir_context *);
        int       (*iterate_shared) (struct file *, struct dir_context *);
        __poll_t  (*poll)           (struct file *, struct poll_table_struct *);
        long      (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long      (*compat_ioctl)   (struct file *, unsigned int, unsigned long);
        int       (*mmap)           (struct file *, struct vm_area_struct *);

    unsigned long mmap_supported_flags;

    int (*open)    (struct inode *, struct file *);
        int (*flush)   (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync)   (struct file *, loff_t , loff_t , int datasync);
        int (*fasync)  (int, struct file *, int);
        int (*lock)    (struct file *, int, struct file_lock *);

    ssize_t       (*sendpage) (struct file *, struct page *, int, size_t , loff_t *,
        ↪ int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
            ↪ long, unsigned long, unsigned long);

        int     (*check_flags)(int);
        int     (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
            ↪ size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
            ↪ size_t, unsigned int);
        int     (*setlease)(struct file *, long, struct file_lock **, void **);
        long    (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif
        ssize_t (*copy_file_range)(struct file *, loff_t , struct file *, loff_t ,
            ↪ size_t, unsigned int);
        loff_t (*remap_file_range)(
        struct file *file_in, loff_t pos_in,
        struct file *file_out, loff_t pos_out,
        loff_t len, unsigned int remap_flags);

        int (*fadvise) (struct file *, loff_t , loff_t , int);
} __randomize_layout;
```

char_device_struct in the linux kernel (github).

```
1  static struct char_device_struct {
2          struct char_device_struct *next;
3          unsigned int major;
4          unsigned int baseminor;
5          int minorct;
6          char name[64];
7          struct cdev *cdev;              /* will die */
8  } *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
9
10 struct cdev {
11         struct kobject kobj;
12         struct module *owner;
13         const struct file_operations *ops;
14         struct list_head list;
15         dev_t dev;
16         unsigned int count;
17 } __randomize_layout;
```

Note: this is a basic example, very interesting!

Block Device I/O

- **Block I/O Subsystem**
  Consists of several layers, with modularised operations (common code ine ach layer).

  The tw main strategies to minimise time accessing block devices:

  1. caching data.
  2. Clustering I/O Operations (store up tasks and execute several at once).

  When given a task, check cache. If data not present, queue request on request queue for device.

- **Direct I/O**
  Bypass the kernel cache when accessing a device.

  Useful for databases and some other applications where caching can reduce performance/-consistency (e.g values very rarely accessed more than once, or doing caching in use level).

50004 - Operating Systems - Lecture 13

Oliver Killane

25/11/21

Lecture Recording

Lecture recording is available here

Linux API

I/O Classes

Character	unstructured	Files and devices
Block	structured	Devices
Pipes	message	Interprocess communication
Socket	message	Network Interface

Sockets

- Allow for bidirectional communication between processes.
- Can be local (e.g processes communicating on the same system), or across the network (e.g connecting to a server) (pipes use machine specific file descriptors, so cannot go over network).
- There are two types (**TCP**, **UDP**)

User Space API

```c
/* Create a file with set permissions. */
fd = create(filename, permission);

/* Open a file/device with mode:
 * 0 - read
 * 1 - write
 * 2 - read/write
 */
fd = open(filename, mode);

/* close a file/device */
close(fd);

/* Attempt to read nbytes from a file/device with descriptor fd to buffer. */
nbytesread = read(fd, buffer, numbytes);

/* Attempt to write nbytes to a file/device with descriptor fd from buffer. */
nbyteswritten = write(fd, buffer, nbytes);

/* Create a pipe */
int fd[2];    /* fd[0] for reading, fd[1] for writing */
pipe(fd);

/* Duplicate the file descriptor. (Uses lowest available) */
newfd = dup(oldfd);

/* Duplicate, using newfd as the new file descriptor. */
newfd = dup2(oldfd, newfd);

/* Issue control command to a device, terminos is an array of control characters */
```

```
31 ioctl(fd, operation, &terminos);
32
33 /* Creates a new special file from a character or block device. */
34 fd = mknod(filename, permission, dev);
```

File Descriptors

- File descriptors are integers, refering to a file opened by the process.
- Process context sensitive (same descriptor, different process → different file)

Each process has 3 descriptors when starting:
0 Standard Input (**stdin**)
1 Standard Output (**stdout**)
2 Standard Error (**stderr**)
By default these point to the terminal which spawned the process.

Blocking & Asynchronous I/O

- **Blocking** process suspended until operation complete
  The I/O call only returns once completed. As a result from the program's perspective this is instantaneous.

  Process making call could be blocked for a long time, we can use threads to get around this (while one blocked, others can run) however this is complex.

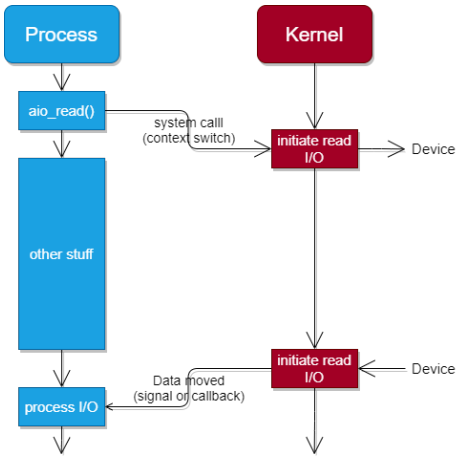
- **Non-Blocking** I/O call returns as much as is available.
  Return immediately (sometimes with data, sometimes with none). By making many calls we can get some data (e.g constantly read until all data is read).

  The provides an application-level polling for I/O (Can constantly request reads from stdin, operating only if there is data at that moment).
  e.g to respond to keystrokes immediately for a terminal game poll stdin by non-blocking reads.

  Turn on using the **fcntl** system call (for sending commands to manage file descriptors, e.g make a file descriptor non-blocking for read/write).

```
1 #include <fcntl.h>
2
3 int fcntl(int fd, int cmd, ... /* arg */ );
```

- **Asynchronous**



- Process runs in parallel with the I/O operations (Non-blocking)
- When operations is complete the process is notified (callback function called, or process signal sent).
- Process can check/wait for I/O completion
- Flexible & Efficient (non-blocking but does not require lots of polling)
- More complex code (e.g futures/promises)
- Potentially less secure if buffers for recieving data are mismanaged.

Security example: Asynchronous read, process sends pointer to buffer to write to. Process then does other stuff, frees buffer, buffer used for something else later. On callback data is written to freed buffer (used after free).

```
1  #include <aio.h>
2
3  int main(int argc, char **argv) {
4      int fd, ret;
5      struct aiocb my_aiocb;
6
7      fd = open("myfile", O_RDONLY );
8
9      /* Allocate buffer for aio request */
10     my_aiocb.aio_buf = malloc(BUFSIZE + 1);
11
12     /* Create aio control structure */
13     my_aiocb.aio_fildes = fd;
14     my_aiocb.aio_nbytes = BUFSIZE;
15     my_aiocb.aio_offset = 0;
16
17     /* Start read request */
18     ret = aio_read(&my_aiocb);
19
20     /* Wait until request is complete (we could also register a signal, or
21      * thread callback instead & do useful work here)
22      */
23     while (aio_error(&my_aiocb) == EINPROGRESS);
24
25         /* Check result from read */
26         if ((ret = aio_return(&my_iocb)) > 0)
27             printf("Successfully read %s", my_aiocb.aio_buf)
28         else
```

```
29              printf("Failed to read :-(")
30  }
```

Responsibilities

- **Computing track, sector & head for disk read** Device Driver (or hardware)
  Requires information about disk layout, in modern HDDs the disk controller is used at only the hardware knows locations of bad blocks.

  (Bad Block - Area that is no longer functioning properly/reliable - physicall damage or corruption)

- **Maintain cache of recently used blocks** Device Independent OS layer
  Useful across many types of block I/O and can be reused/shared between different devices

- **Write Commands to Drive Registers** Interrupt Handler
  If time-critical can be performed in an interrupt routine, if it requires more time (or is device specific) can be done by device driver.

- **Checking user is allowed to use a device** Device Independent OS layer
  Applicable to many different types of devices, OS can enforce policy uniformly over all devices.

- **Converting integers to ASCII for printing** User-Level I/O Layer
  Data representations managed by a library users link against (e.g for pretty printing), this gives user programs lots of flexibility about how to convert.

  Aslo more performant as requires no kernel involvement (which would require context switches).

Disk Management

History

1956 - IBM 305 RAMAC (First Commercial Hard Disk)

- 4.4MB
- $1.5m^2$
- \$160,000

$\longrightarrow$

2005 - Toshiba 0.85" disk (Smallest ever)

- 4GB
- $<$ \$300

Disk capacity has grown exponentially, through access speeds have not kept pace.

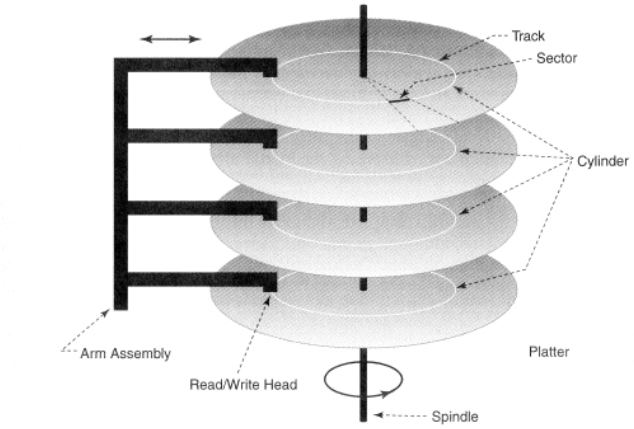
Recently this has been driven by demand from cloud providers & services (e.g Youtube, microsoft 365, facebook etc).

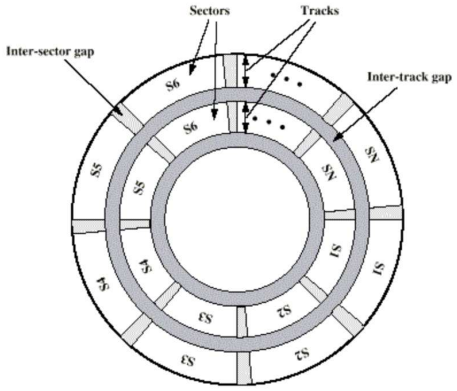
HAMR

Heat Assisted Magnetic Recording. To ensure high rpecision, even when working at very high speeds, the platter is such that it can only be written to when heated.

A laser shines just ahead of the read/write head, heating areas that will be written too. Cold areas ignore any reading/writing.

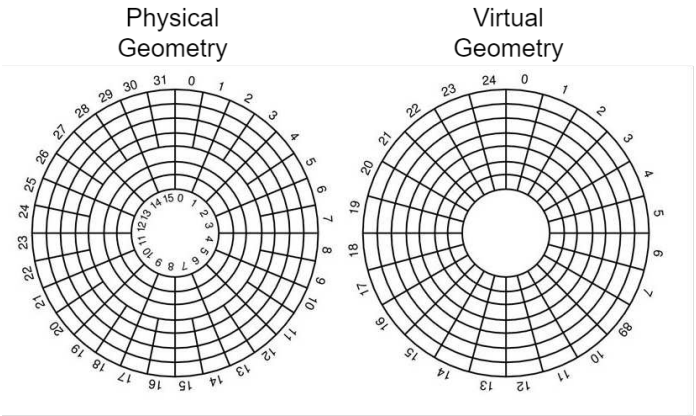
Disk Layout





(Note that **cylinder** refers to all the tracks on top opf eachother / same number on each disk. Innermost cylinder is all the innermost tracks on all the platters)

Sector Layout



- Surface split into tracks (concentric circles)
- Track split into equal size sectors (outer tracks have more sectors)

- Physical address is (cylinder, surface, sector), but this is hidden from the OS.

Modern systems use **logical sector addressing/logical block addresses (LBA)**:

- Sector Numbered consecutively $0..n$
- Makes disk management much easier (just get sector number & offset in sector)
- Helps to work around BIOS limitations (e.g IBM PC BIOS could only address 8GB (6 bits sector, 4 head, 14 cylinder), this is too small).

Capacity

Some sources will use 1000, others 1024 as base for KB, MB, GB, TB. Ensure you are consistent with which base you use during exams.

Disk Formatting

Low Level Format

- **Preamble** Identify the type of block
- **Data** For storage
- **ECC** Error Correction Codes (to correct small errors in read/write)

Some techniques are also used such as **interleaving** (for older systems, sequential data spread apart such that CPU has time to process a sector, before the next one is read) and cylinder skew (cylinder skew of n means when the head is at sector x on one surface, on the next its at sector $x + n$).

High Level Format

- **Boot Block** A block (usually at start of disk) that is dedicated to starting the system.
- **Free Block List** Stores which blocks are currently in use, and which are free to allocate.
- **Root Directory** Start of the file system, all subdirectries emenate from here.
- **Empty File System**

Example Question

A disk controller with enough memory can perform **read-ahead**, reading blocks before the CPU has requested them.

Should it also do **write-behind** (writing to controller memory, informing CPU it is done, only to write back later)

Answer: Not in general - as disk controller memory is still volatile, so in a power loss the data would be lost. (Method around is a small battery large enough to write back in case of a power failure).

Disk Delay

Sector Size	521 bytes	
Seek time	adjacent cylinder (<1ms), avg (8ms)	(Move to correct track)
Latency/Rotation time	$4ms$	(Spin platter to get to beginning of block)
Transfer time	$> 100MB/s$	(Spin platter over block being read)

Seek time is $2-3$x larger than latency time

Disk Scheduling

- Minimise seek/latency times
- Order pending requests to take advantage of head position

Disk Performance

Where:

- b - bytes to transfer
- N - bytes per track
- r - rotation speed in r/s (rotations per second)

- **Seek Time** t_{seek}
- **Transfer Time** $t_{transfer} = \dfrac{b}{r \times N}$

  $\dfrac{1}{r}$ is seconds per revolution.
- **Total Access Time** $t_{access} = t_{seek} + \dfrac{1}{2 \times r} + \dfrac{b}{r \times N}$

  Seek + latency time (to get to start average case rotate half → disk typically only rotates in one direction) + How much to rotate while reading (e.g reading x bytes from track size $2x$ requires a half rotation)

50004 - Operating Systems - Lecture 14

Oliver Killane

18/12/21

Lecture Recording

Lecture recording is available here

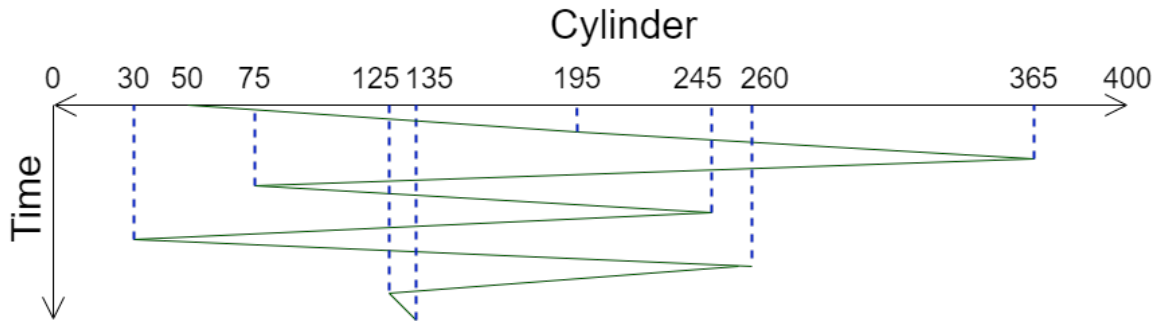
Disk Scheduling

(FCFS) First Come First Served

Requests completed in the order they were received.

- For light loads this is fine (time between requests is much larger than time taken to fulfil any request).
- Low performance for heavy loads (ends up traversing the disk more than optimal to get each request).
- Fair (no bais towards any process).

For example:
Head at 50. Queue: $195, 365, 75, 245, 30, 260, 125, 135$
Operations: $50 \rightarrow 195 \rightarrow 365 \rightarrow 75 \rightarrow 245 \rightarrow 30 \rightarrow 260 \rightarrow 125 \rightarrow 135$

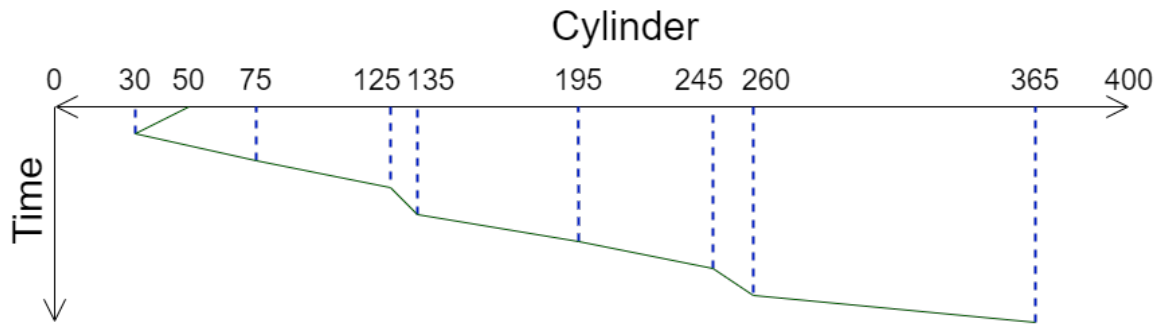


(SSTF) Shortest Seek Time First

Order requests according to shortest seek distance from the current head position

- Baised against innermost and outermost tracks (middle tracks on average closer, even with random head positions)
- Unpredictable performance (Requests may face a long delay if several very close requests are serviced)
- Processes can use dummy requests to keep control of head and have their requests prioritised.
- Can delay requests indefinitely (e.g many requests come in very close to eachother, trapping the head).

Head at 50. Queue: $195, 365, 75, 245, 30, 260, 125, 135$
Operations: $50 \rightarrow 30 \rightarrow 75 \rightarrow 125 \rightarrow 135 \rightarrow 195 \rightarrow 245 \rightarrow 260 \rightarrow 365$

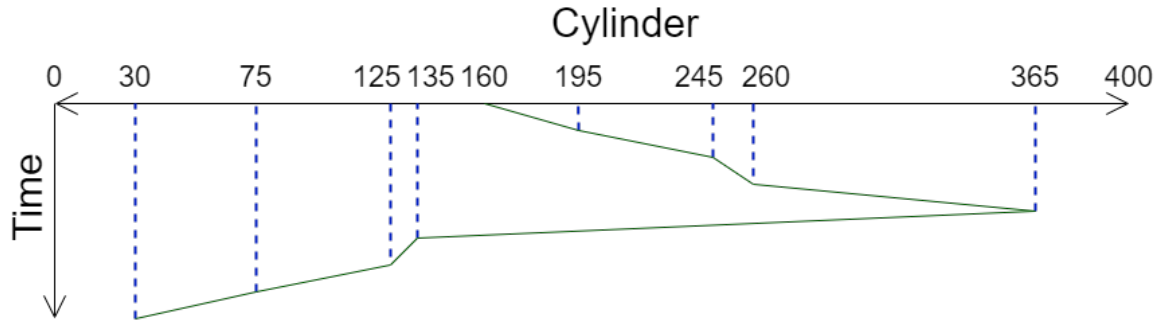


SCAN Scheduling

Also called elevator scheduling. Select requests with the shortest seek time in the preferred direction.

- Only change direction when at the outermost/innermost cylinder (no more requests in the direction)
- Long delays for requests that are not in the path of the algorithm (e.g come in just behind the head) and for the extremes.
- Base for the most common algorithms used.
- Suffers from same delay issue as **SSTF**, though reduced (only in one direction)

Head at 160. Queue: $195, 365, 75, 245, 30, 260, 125, 135$
Operations: $160 \rightarrow 195 \rightarrow 240 \rightarrow 260 \rightarrow 365 \rightarrow 135 \rightarrow 125 \rightarrow 75 \rightarrow 30$

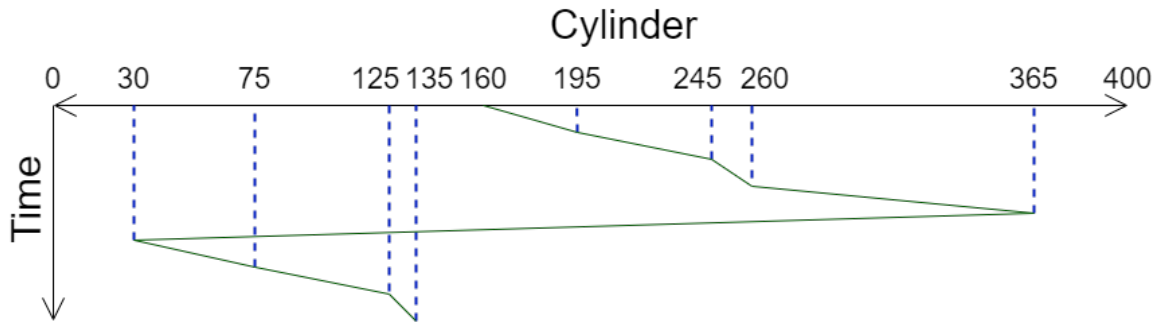


C-SCAN Scheduling

Much like **SCAN** but scanning in one direction only, jumping to the start (e.g innermost) when at the end (e.g outermost)

- Lower variance of requests on extreme tracks
- Largely reduces issue of indefinite wait from **SSTF**.

Head at 160. Queue: $195, 365, 75, 245, 30, 260, 125, 135$
Operations: $160 \rightarrow 195 \rightarrow 240 \rightarrow 260 \rightarrow 365 \rightarrow 30 \rightarrow 75 \rightarrow 125 \rightarrow 135$

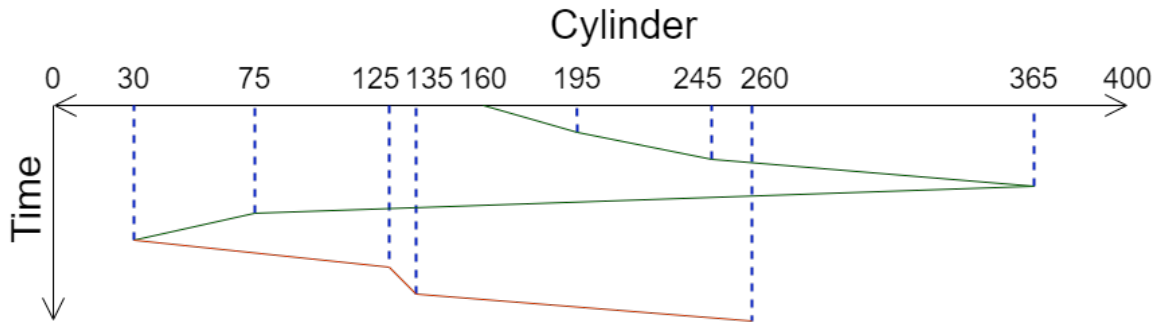


N-Step SCAN Scheduling

Same as **SCAN**, however only services requests waiting when the sweep began (for each sweep)

- Requests arriving during sweep are serviced before end of the sweep (no long waits)
- No indefinite waits possible.

Head at 160. Queue: $195, 365, 75, 245, 30$ Then $260, 125, 135$
Operations: $160 \rightarrow 195 \rightarrow 245 \rightarrow 365 \rightarrow 75 \rightarrow 30 \rightarrow (\textit{new sweep})\ 125 \rightarrow 135 \rightarrow 260$



Linux Disk Scheduling

struct bio

Can be found here in the linux kernel.

```
/* main unit of I/O for the block layer and lower layers (ie drivers and
 * stacking drivers)
 */
struct bio {
        struct bio          *bi_next;   /* request queue link */
        struct block_device     *bi_bdev;
        unsigned int         bi_opf;    /* bottom bits req flags, top bits REQ_OP.
                                         * Use accessors.
                                         */
        unsigned short       bi_flags;  /* BIO_* below */
        unsigned short       bi_ioprio;
        unsigned short       bi_write_hint;
        blk_status_t         bi_status;
        atomic_t             __bi_remaining;
        struct bvec_iter     bi_iter;
        bio_end_io_t        *bi_end_io;
        void                *bi_private;
#ifdef CONFIG_BLK_CGROUP

        /* Represents the association of the css and request_queue for the bio.
         * If a bio goes direct to device, it will not have a blkg as it will
         * not have a request_queue associated with it.  The reference is put
         * on release of the bio.
         */
        struct blkcg_gq         *bi_blkg;
        struct bio_issue         bi_issue;
#ifdef CONFIG_BLK_CGROUP_IOCOST
        u64                      bi_iocost_cost;
#endif
#endif

#ifdef CONFIG_BLK_INLINE_ENCRYPTION
        struct bio_crypt_ctx    *bi_crypt_context;
#endif

        union {
#if defined(CONFIG_BLK_DEV_INTEGRITY)
                struct bio_integrity_payload *bi_integrity; /* data integrity */
#endif
        };

        unsigned short bi_vcnt; /* how many bio_vec's */

        /* Everything starting with bi_max_vecs will be preserved by bio_reset() */
        unsigned short  bi_max_vecs;    /* max bvl_vecs we can hold */
    atomic_t        __bi_cnt;   /* pin count */
        struct bio_vec *bi_io_vec;      /* the actual vec list */
        struct bio_set *bi_pool;   4

        /* We can inline a number of vecs at the end of the bio, to avoid
         * double allocations for a small number of bio_vecs. This member
         * MUST obviously be kept at the very end of the bio.
         */
        struct bio_vec          bi_inline_vecs[];
};
```

- **I/O Requests added to request list**
  There is one request list per device, a **bio** that keeps track of the pages associated with the request.
- **Block Device drivers define a request operation for the kernel to use**
  Interface provided by function pointers.
  - Kernel passes ordered request list to driver.
  - Driver completes all requests in the list.
  - Drivers use the read/write operations defined by the kernel (uniform interface fro kernel to send uniform request types to many different drivers).
- **Driver Based ordering**
  Some drivers bypass kernel ordering and do it themselves (e.g **RAID**). This is done for more complex disk setups where assumptions made by the kernel ordering algorithm do not apply.
- **Default Algorithm: SCAN Scheduling**
  Kernel attempts to merge requests to adjacent blocks (grouping adjacent requests results in less seek time, higher request throughput)

  However read requests may starve during very large writes (from merging), if these are done synchronously, a program may hang.
- **Deadline Scheduler**
  Ensures that reads are performed before a set deadline to prevent long or indefinite waits. (Elimintates read request starvation)
- **Anticipatory Scheduler**
  Delay after read requests completed. If a process sends a second synchronous read request, it will be attended to quickly (due to delay after completing the first).
  - Reduces Excessive seeking (Second request will likely be very close to the first, so avoid seeking away, and then back)
  - Can reduce throughput (if no more read requests are sent by the process during the delay, or if the request sent requires a large seek)

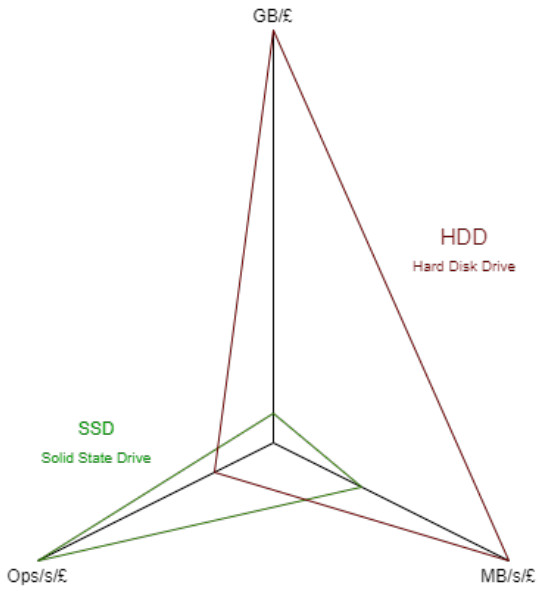
Solid State Drives

SSD Scheduling

Scheduling for SSDs do not require scheduling algorithms as many memory modules can be read/written in parallel and write/read speed is approximately constant.

However drivers need to overcome issues with limited writes, tracking virtual to physical blocks & assignment of free blocks.

- Very high bandwidth (e.g 1GB/s SSD vs 100MB/s HDD)
- Lower latency
- High parallelism



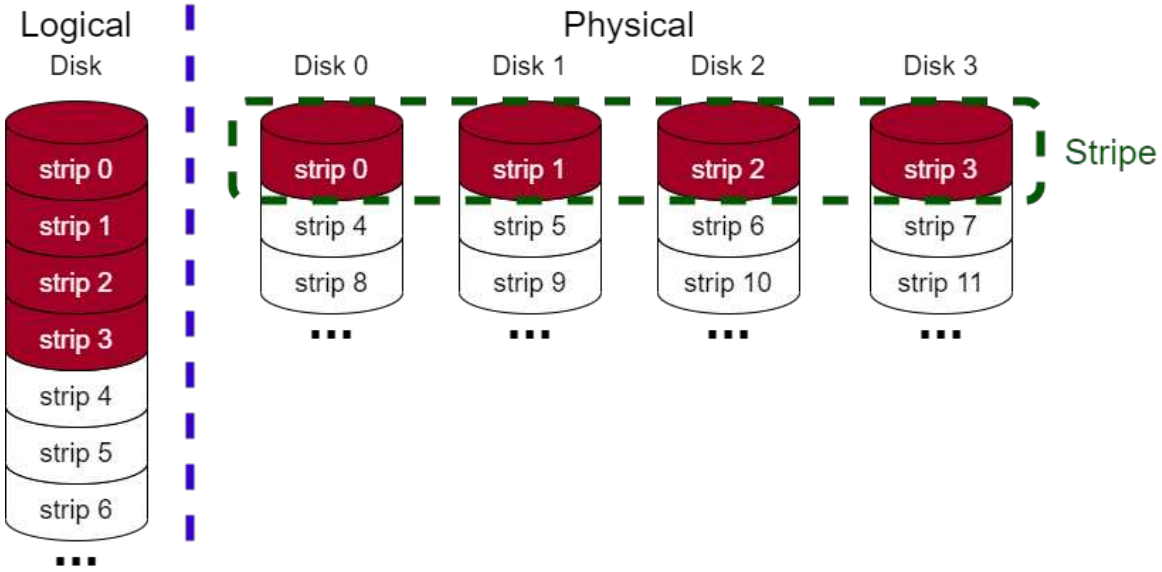
RAID

Disk performance has not kept pace with CPU performance, creating a bottleneck. Redundant Array of Inexpensive Disks (**RAID**) increases disk based system performance by using many disks in parallel.

- An array fo physical drives appears as a single virtual drive.
- Distributes stored data over physical disks to allow parallel operation, improving performance.
- Use redundant disk capacity to respond to disk failure (more disks means lower mean-time-to-failure, more disks, higher chance a single disk in the group fails).

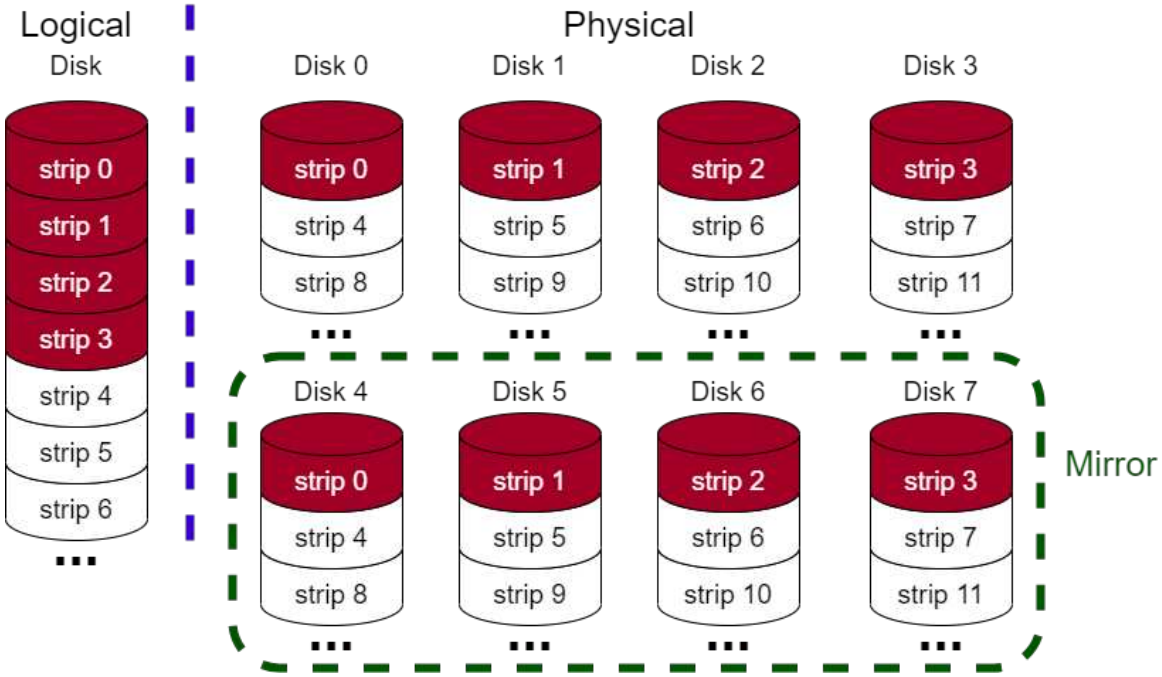
There are several levels of **RAID**, each with differing performance, redundancy and space-efficiency/cost.

RAID 0 - Striping



- Spread blocks in round robin fashion across disks.
- Can concurrently seek/transfer data (provided blocks on different physical disks).
- Can balance load across disks (sometimes).
- No redundancy (any disk failure will result in data being lost).

RAID 1 - Mirroring



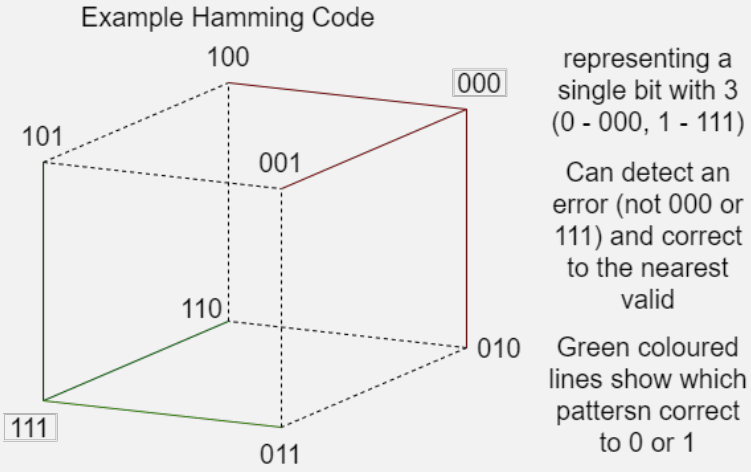
Mirror Data across disks to increase redundancy.

- Reads can be serviced by either disk (e.g can read $0, 1, 2, 3, 4, 5, 6, 7$ at the same time).
- Writes must update both disks in parallel (e.g can effectively only write to 4 disks at a time). (Slower)
- Failure recovery is easy (when a disk fails, use its mirror).
- Low space efficiency and hence high cost (store everything twice).

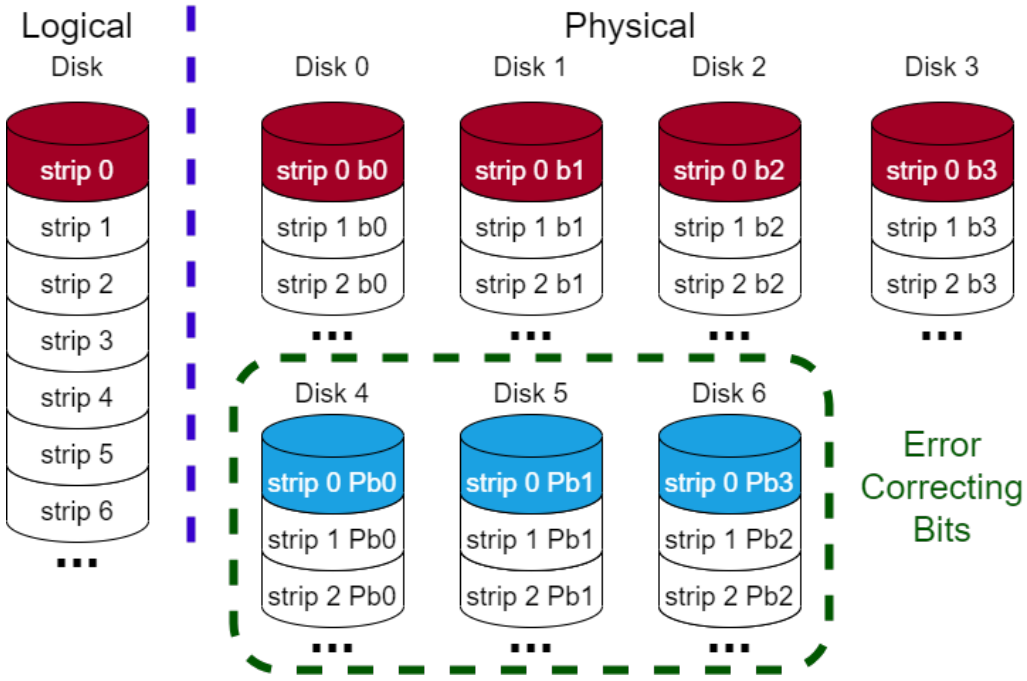
RAID 2 - Bit-Level Hamming

Hamming Codes

A family of error correcting codes that make use of hamming distance (number of bits different between two patterns) to correct single bit errors.



More complex codes can be used to check larger bit patterns with very few bits. The general algorithms can be seen here

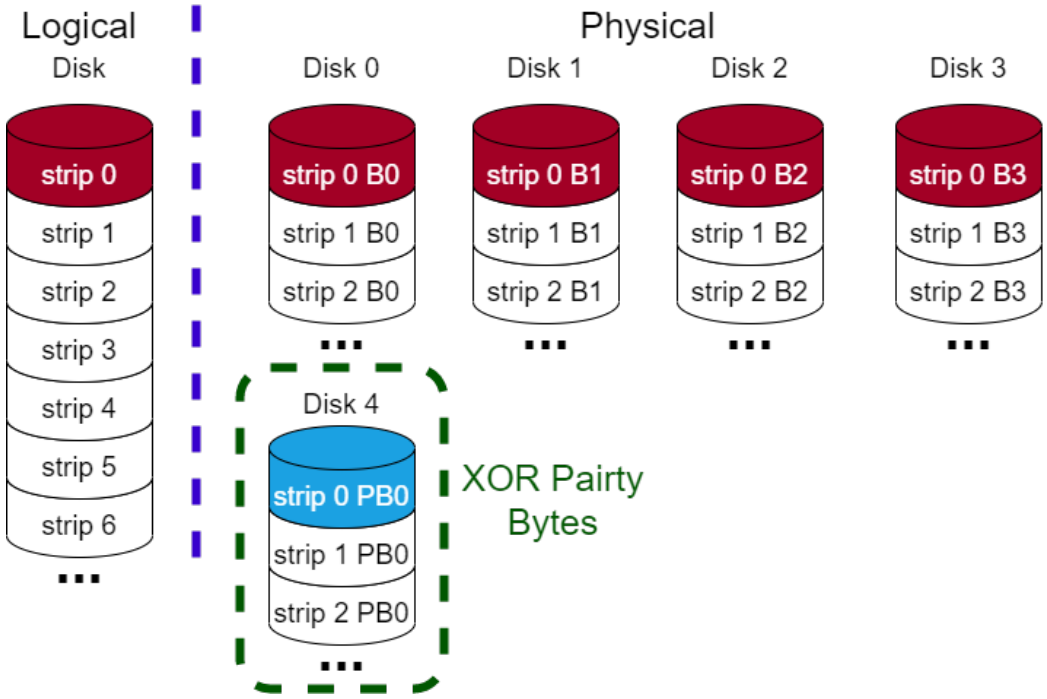


Parallel access by striping at the bit level.

- Consecutive (in this case 4) bits are read in parallel, hence very high throughput (always reading/writing in parallel).
- Hamming error-correcting codes used to detect and correct single bit errors (can detect but not correct double bit errors).
- Cannot process requests in parallel (each request requires all disks, no concurrency).
- High storage overhead (less space efficient, increasing cost).
- Large number of writes as the error correcting codes must be updated (can become a bottle-neck) & every disk must write for every write operation.

RAID 3 - Byte-Level XOR

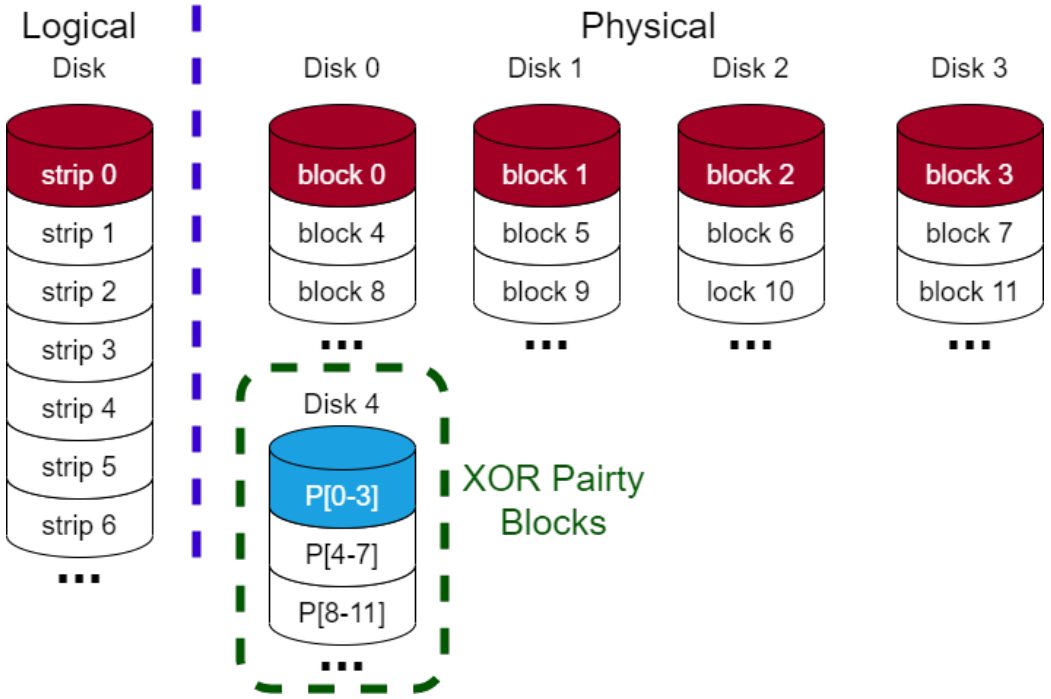
A single parity strip used (data on other disks XORed together).



Can reconstruct missing data from parity and remaining data when a disk fails (disk reports).

- Easy to reconstruct data when a disk fails
- More space efficient than **RAID 2**
- Only one I/O request at a time (but each request can be read in parallel from all disks)

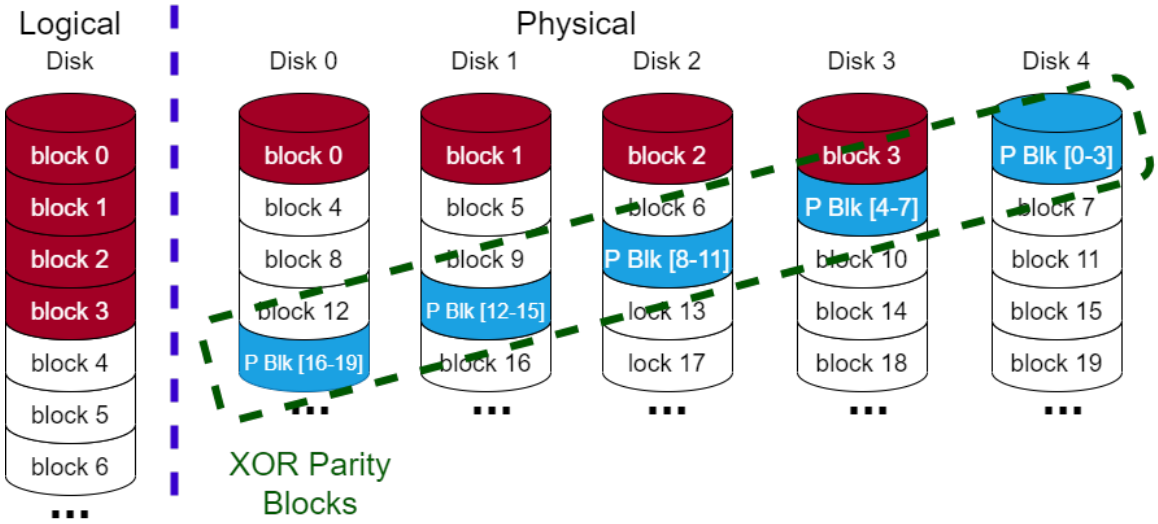
RAID 4 - Block Level XOR



Parity Strip is XOR over blocks, allowing entire blocks to be accessed independently on different disks.

- Allows read requests to be serviced concurrently.
- Low redundancy overhead.
- Parity must be updated after every single write, Parity disk becomes a bottlenecks

RAID 5 - Block Level Distributed XOR



The most commonly used **RAID** level, by distributing parity there is potential for concurrency.

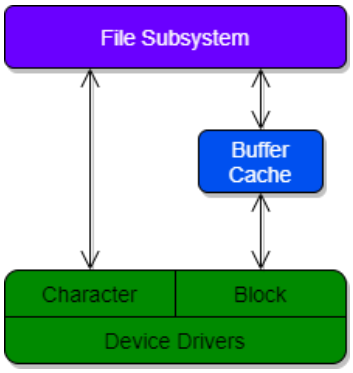
- Some potential for write concurrency as parities are on different disks.
- Good storage/efficiency tradeoff.
- Reconstruction of disk is non-trivial (and slow).

RAID Level summary

Speeds compared with using a single disk.

Category	Level	Description	I/O Data Transfer		I/O Request Rate	
			Read	Write	Read	Write
Striping	0	Non-Redundant	↑	↑	↑	↑
Mirroring	1	Mirrored	↑	=	↑	=
Parallel Access	2	Redundant (Hamming ECC)	↑↑	↑↑	=	=
	3	Redundant (Bit Interleaved Parity)	↑↑	↑↑	=	=
Independent Access	4	Block interleaved Parity	↑	↓	↑	↓
	5	Block interleaved distributed parity	↑	↓	↑	= or ↓

Disk Caching



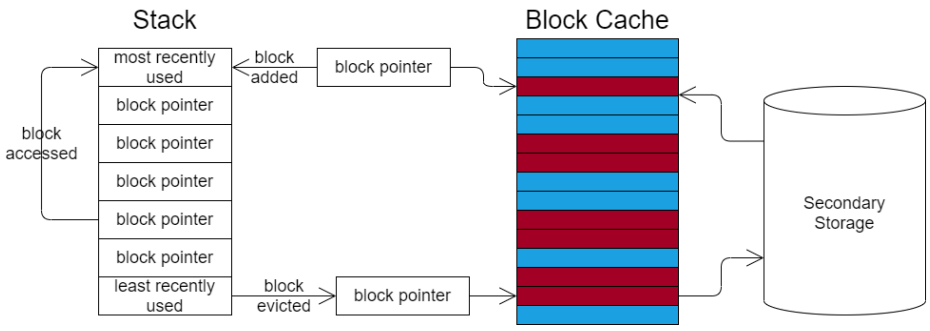
We can cache sectors of disk in main memory to reduce access times.

- Buffer contains copies of disk sectors.
- OS manages disk in terms of **blocks** which are likely much larger than sectors (so loads multiple sectors).
- Must ensure contents are saved in case of failure (e.g lazy writing is very complex).
- Cahce has finite space, so a replacement policy must be implemented.

(LRU) Least Recently Used

Replace the block that was in cache longest with no references.

Cache is a stack of pointers to blocks in memory, when a block is referenced, it is pushed to the top of the stack. Replacement evicts the block at the bottom of the stack.

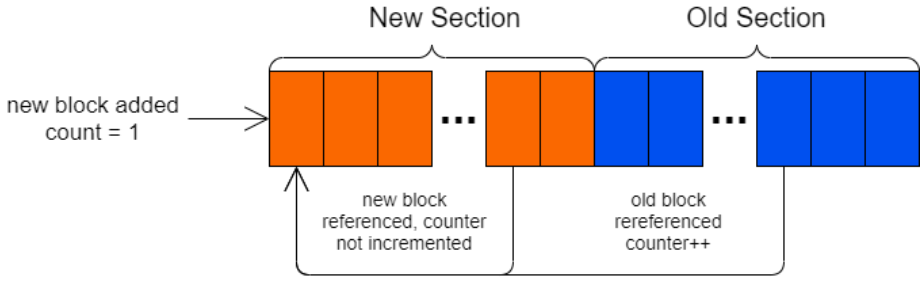


This replacement policy does not track how many times a block is accessed, only the relative time of the last access (through stack position).

(LFU) Least Frequently Used

- Replace the block with the fewest references.
- Each block has a counter (incremented when referenced), the lowest counter block is evicted during replacement.
- To avoid this, can use frequency based replacement.

Frequency-Based Replacement



To prevent items that get a sudden burst of accesses from lingering for a long time in the cache, we only increment the count when a block is shuffled out of the **new** section.

File Systems

File systems organise information. Their main objectives:

- **Non-volatile, long term storage**
- **Sharing information** e.g compilers, applications, text data etc.
- **Concurrent Access** Many programs can access many files pesudo-simultaneously
- **Convenient Organisation** symbolic names, directories
- **Easy management of data** automatic backup, snapshots etc.
- **Security** Permissions, read/write, hidden files

File Naming

A **file** is an arbitrary size collection of data, extensions allow for easy identification of type from an identifier:

Type	Extension	Function
executable	exe,com,bin,no extension	read & load to run machine code program.
object	obj, o	compiled machine language, not linked.
source code	c, cpp, java, rs, py, s	Source code, extension identifies language.
batch/bash	bat (windows), sh (unix)	Script of commands to be interpreted by the terminal.
text	txt	Text data (usually **ascii**).
library	lib, a, so, dll	Libraries that programs can link to.
archive	arc, zip, tar	Compressed archives (for transmission or storage).

There are also several types of file.

Type	Description
Hard Links	A file that aliases the data of another file (refers to the specific location of the data).
Soft Links	A soft/symbolic link aliases the path to a file by another (e.g file points to another file, which in turn points to its data). The **ln** command can be used to soft link files in unix based systems.
Regular	Normal file as discussed in the table above.
Directory	A collection of files, that can itself be added to directories.
Character Special	A file that provides access to a character I/O device
Block Special	Same as a character special file, but for block devices.

Filesystem support functions

- **Name Translation** Converting paths to disks & blocks (logical) for use by the driver.
- **Management of Disk Space** Allocating and deallocating storage for files.
- **File locking for exclusive access** Important when many programs may access a fiule concurrently (can fuse the **fcntl** syscall with the **F_SETLK**,**F_SETLKW**, and **F_GETLK**)
- **Performance Optimisation** Caching/Buffering
- **Protection against system failure** Backup/Restore in case of a crash/power failure/unexpected shutdown
- **Security** Enforcing file permissions

File Attributes

- **Basic**
  - **File Name** Symbolic name, unique in directory
  - **File Type** e.g text, binary, executable, directory
  - **File Organisation** placement of contents in blocks (sequential, random)
  - **File Creator** program that created the file
- **Address information**
  - **Volume** Disk drive, partition
  - **Start Address** cylinder, head, sector (logical block addressing information)
  - **Size Used**
  - **Size Allocated**
- **Access Control Information**
  - **Owner** User that controls the file
  - **Authentication** Locked files may need a password/key
  - **permitted actions** e.g read/write, dlete permissions (owner/others)
- **Usage Information** Metadata for paper-trail.
  - **Creation Timestamp** Date & Time
  - **Last Modified** (can include the user that made the modification)
  - **Last Read**
  - **Last Archived** For keeping track of backups
  - **Expiry Date** For automatic deletion (e.g recycle bin contents)
  - **Access activity** Metadata for reads/writes etc. (can be used in improving performance)

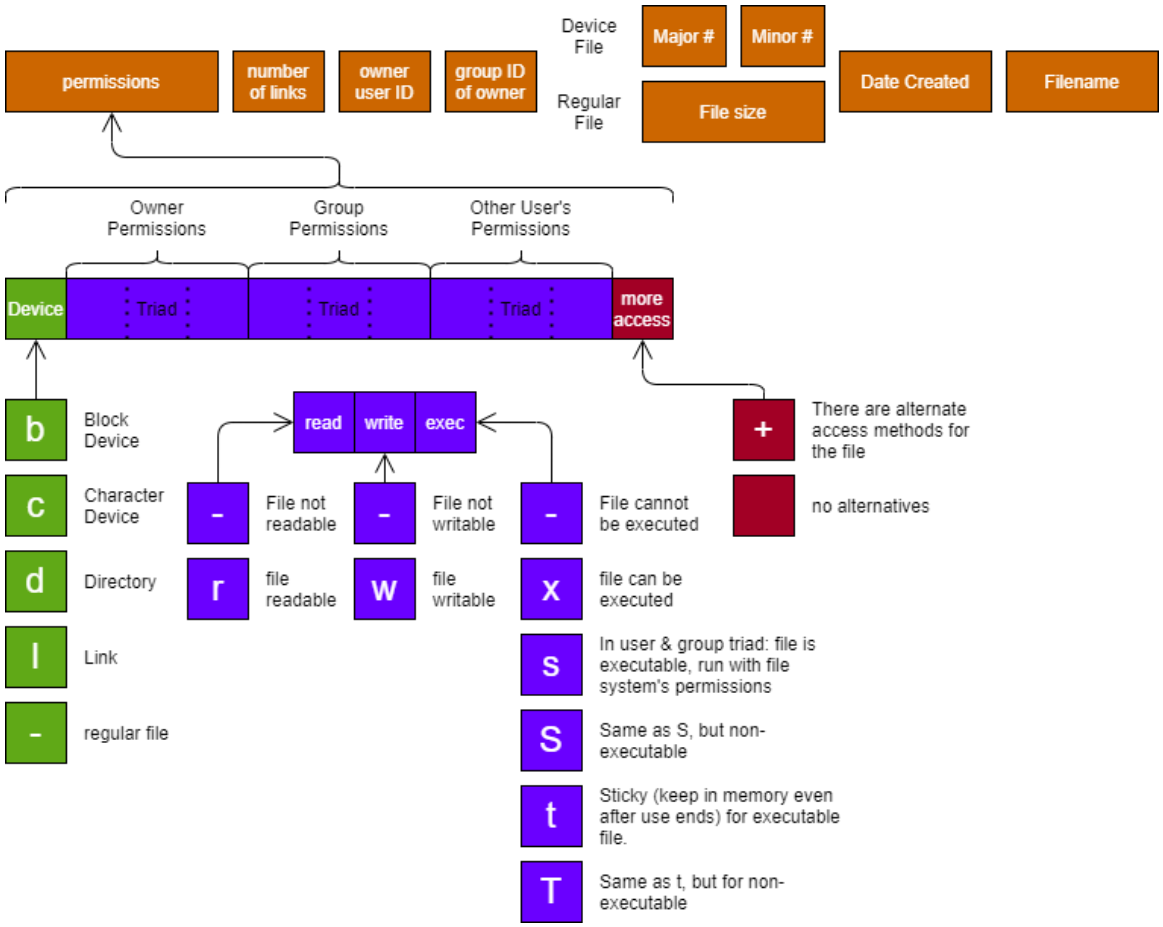
Unix/Linux Files

Common Filesystem Syscalls

```
1  /* UNISTD functions (header for POSIX api) */
2  #include <unistd.h>
3  /* Close a file */
4  int close(int fd);
5
6  /* Read COUNT bytes from file FD into buffer BUF. Return number of bytes read. */
7  ssize_t read(int fd, void *buf, size_t count);
8
9  /* Write to file FD, COUNT bytes from BUF. Return number of bytes written. */
10 ssize_t write(int fd, const void *buf, size_t count);
11
```

```c
/* Seek to the given OFFSET in file FD, options in WHENCE. "lseek" stands for
 * long seek (offset is 64 bits for large files).
 */
off_t lseek(int fd, off_t offset, int whence);

/* Return file information for file at PATHNAME into buffer STATBUF. */
int stat(const char *restrict pathname, struct stat *restrict statbuf);

struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
     *  precision for the following timestamp fields.
     *  For the details before Linux 2.6, see NOTES.
     */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */

    #define st_atime st_atim.tv_sec      /* Backward compatibility */
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec
};


/* File Control functions */
#include <fcntl.h>

/* Note that interestingly, despite close being unistd, open is not. This is as
 * many of the flags required for open are declared in FCNTL, and the developers
 * wanted to avoid polluting UNISTD with these.
 */

/* Open a file at PATHNAME with FLAGS to get a file descriptor. */
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

/* File controls (e.g lock, set read status). */
int fcntl(int fd, int cmd, ... /* arg */ );
```

File Attributes




```
nolinks# For a device:
# <permission attributes> <no. links> <owner> <group> <major> <minor> <creation date>
    ↪ <filename>

crw------- 1 root root    5,    1 Dec 20 14:59 console
# Character device, readable and writeable, but not executable by the owner

brw------- 1 root root    1,   15 Dec 20 14:59 ram15
# Block device, readable and writeable, but not executable by the owner

# For a file:
# <permission attributes> <no. links> <owner> <group> <size> <creation date> <
    ↪ filename>

-rwxrwxrwx 1 oliverkillane oliverkillane  21775 Dec 20 16:31 'Lecture 14.tex'
# File, readable, writable and can be executed by any user
```

File System Organisation

Space Allocation

File size is variable (can increase or decrease) and is allocated on the disk in blocks (usually $512 \rightarrow 8192$ bytes). The block size is determined by the file system.

Block Size too Small

- High overhead for managing large files (large number of blocks to keep track of)
- High file transfer time (large file $\rightarrow$ lots of blocks across the disk $\rightarrow$ lots of seeking back and forth)

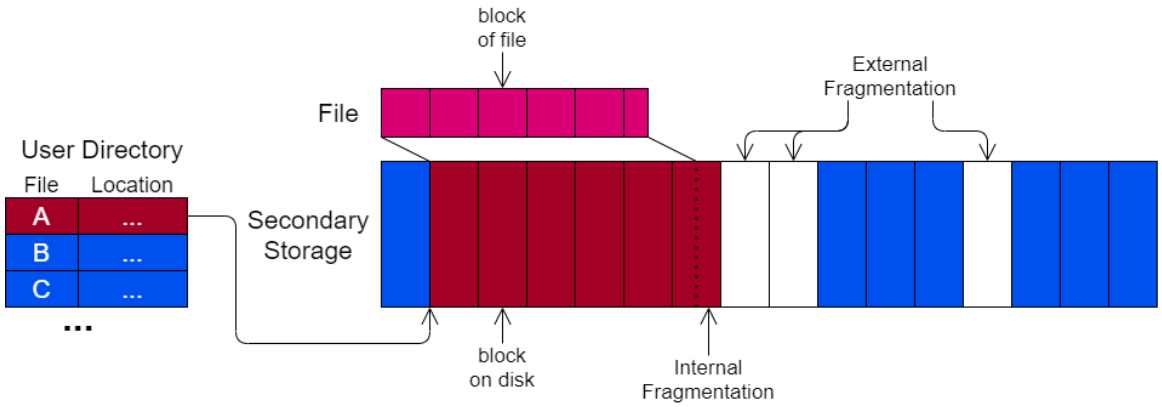
Block Size too Large

- Internal fragmentation (Files leave parts of large blocks unused - wasteful)
- Small files waste lots of space.
- Caching is based on blocks, so end up having a very large cache space, or unable to cache many blocks.

Contiguous File Allocation

Place file data on a contiguous stretch of addresses on storage device. Similar to segment based memory management.

- Successive logical records are usually physically adjacent, reducing seek time when reading the file (seek to start, then just traverse file).
- External Fragmentation can occur (unused blocks between files).
- If unable to allocate new blocks as a file grows, must transfer to new large free section (expensive, requires lots of I/O).

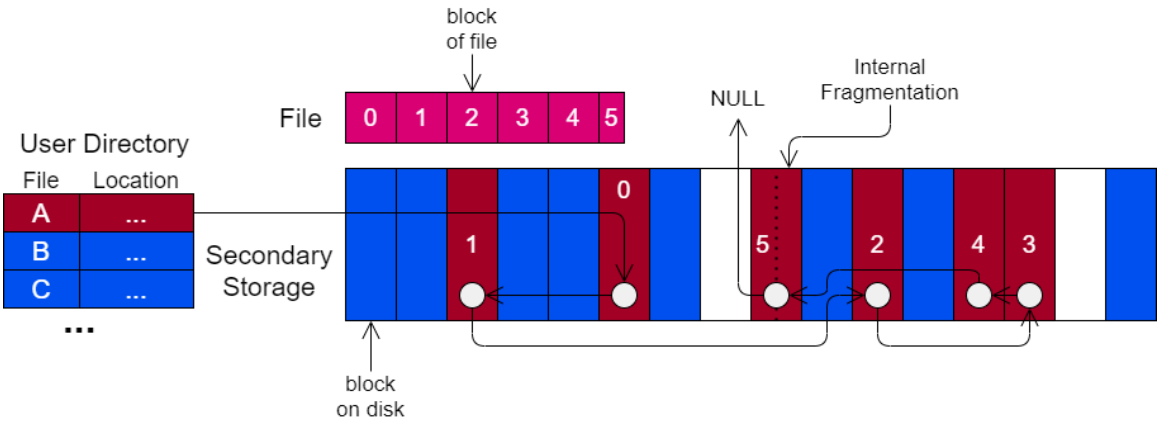


Block Linkage/Chaining

Each file contains a linked list of blocks. When locating a block, traverse the linked list to by seeking to block, reading pointer.

- Can grow files without expensive re-allocation (just set pointer of end block to point to a newly allocated one).
- No external fragmentation (as all blocks can be used in any order, by any file).
- Space used for pointer in each block.
- Traversing files requires lots of seeking from block to block.

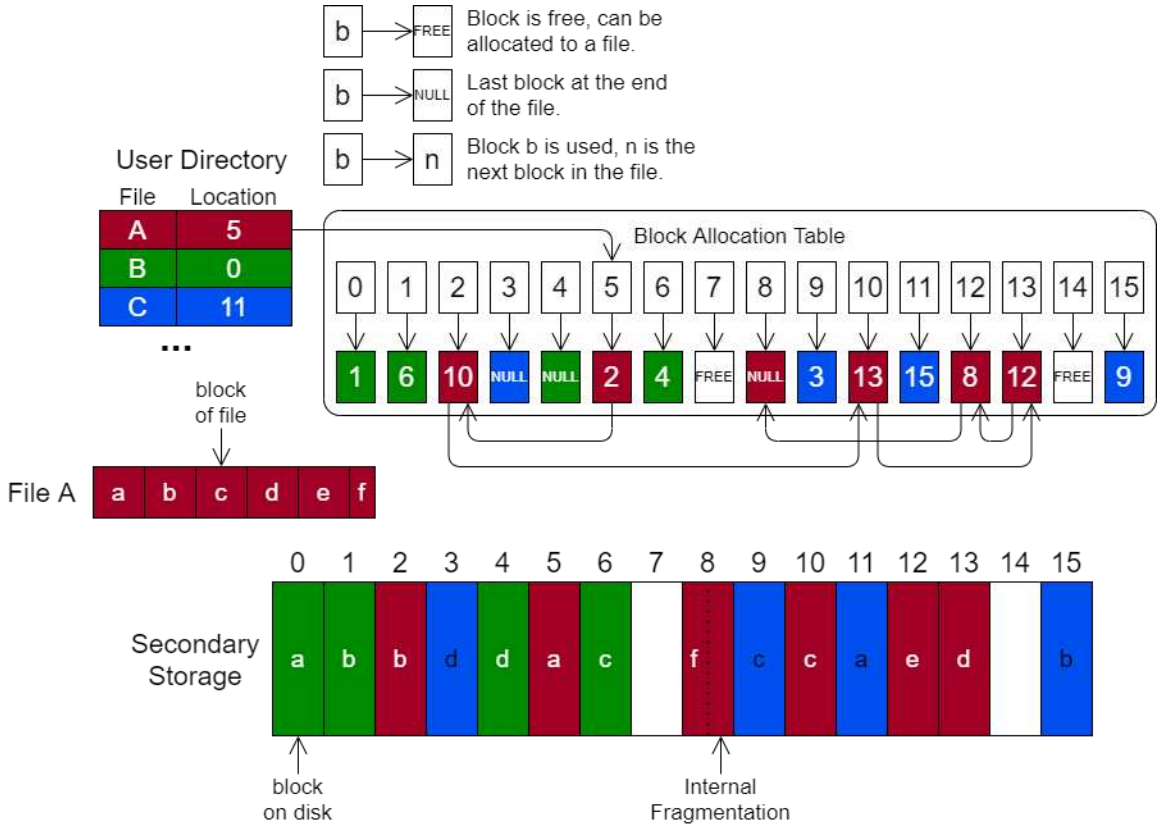
- For small block sizes the number of seeks to traverse a file increases.



Block Allocation Table

Uses a directory mapping files to first block. Table maps blocks to the next block in the file, indicating free spots (**FREE**) and block with no next (end of file - **NULL**).

- File allocation table can be cached in memory for fast lookup.
- Does not require lengthy seeks to traverse the block numbers for the file.
- No external fragmentation.
- However files can become fragmented (spread across disk) which reduces read/write speed (e.g must do lots of seeks to read the whole file), so should be periodically defragmented (disk reorganised to place files blocks next to eachother, very expensive operation).
- Table can become impractically large, using up lots of memory, and more than one block of storage.

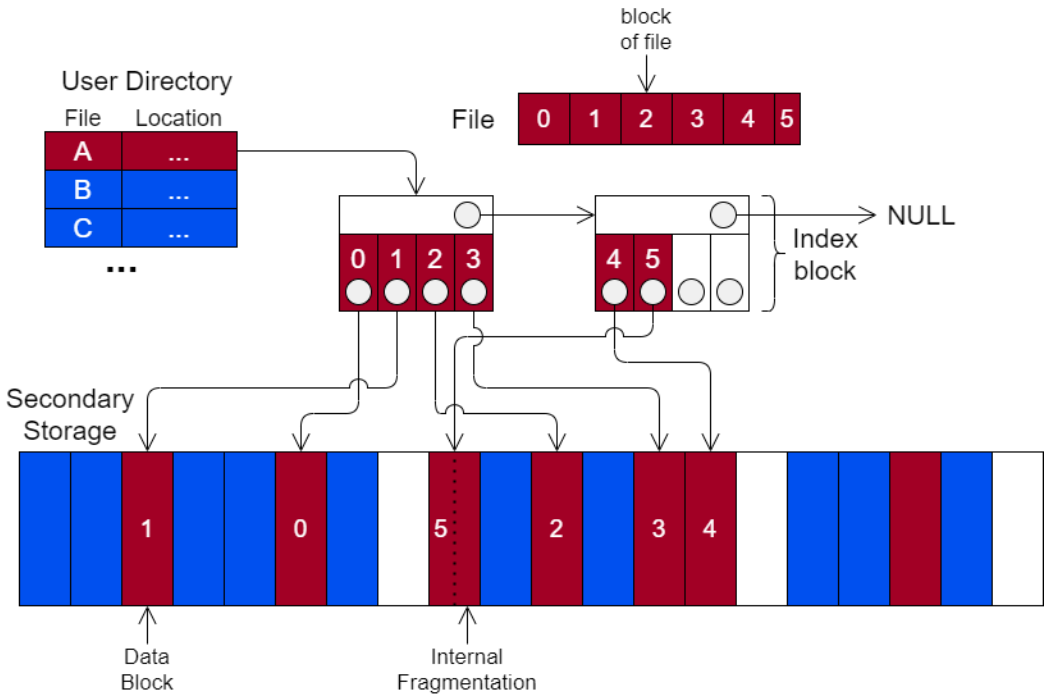


This system is used for **microsoft**'s **FAT16/32** file system (with the table cached in memory).

Index Blocks

Each table has one or more index blocks. Index blocks contain an array of pointers to data blocks, and pointers to subsequent index blocks. Basically page tables for file systems. The example below has miniscule index blocks for illustration purposes.

- Can search through index blocks to find location fo file blocks easily.
- No external fragmentation & can extend files easily.
- Index blocks can be cached in memory just like data blocks.
- Index blocks are per-file, hence can load file's table, rather than have an enormous global table as with **FAT**



Linux Inodes

Linux/Unix uses the **Index Blocks** strategy through **inodes** (index nodes).

Inode contain:

- Type & Access control.
- Number of links to that inode.
- User & Group ID.
- Access & Modification time.
- Inode change time (e.g when permissions, data stored in inode were changed).
- Direct, Indirect, Double Indirect & Triple Indirect pointers to data blocks.

The inode struct can be found here in the linux kernel.

```
1  struct inode {
2          umode_t         i_mode;
3          unsigned short  i_opflags;
4          kuid_t          i_uid;
5          kgid_t          i_gid;
6          unsigned int    i_flags;
7
8  #ifdef CONFIG_FS_POSIX_ACL
9          struct posix_acl        *i_acl;
10         struct posix_acl        *i_default_acl;
11 #endif
12
13         const struct inode_operations   *i_op;
14         struct super_block              *i_sb;
15         struct address_space            *i_mapping;
```

```
16
17  #ifdef CONFIG_SECURITY
18          void                    *i_security;
19  #endif
20
21          /* Stat data, not accessed from path walking */
22          unsigned long           i_ino;
23          /*
24           * Filesystems may only read i_nlink directly.  They shall use the
25           * following functions for modification:
26           *
27           *    (set|clear|inc|drop)_nlink
28           *    inode_(inc|dec)_link_count
29           */
30          union {
31                  const unsigned int i_nlink;
32                  unsigned int __i_nlink;
33          };
34          dev_t                   i_rdev;
35          loff_t                  i_size;
36          struct timespec64       i_atime;
37          struct timespec64       i_mtime;
38          struct timespec64       i_ctime;
39          spinlock_t          i_lock;     /* i_blocks, i_bytes, maybe i_size */
40          unsigned short      i_bytes;
41          u8                  i_blkbits;
42          u8                  i_write_hint;
43          blkcnt_t            i_blocks;
44
45  #ifdef __NEED_I_SIZE_ORDERED
46          seqcount_t              i_size_seqcount;
47  #endif
48
49          /* Misc */
50          unsigned long       i_state;
51          struct rw_semaphore i_rwsem;
52
53          unsigned long           dirtied_when;   /* jiffies of first dirtying */
54          unsigned long           dirtied_time_when;
55
56          struct hlist_node i_hash;
57          struct list_head  i_io_list;    /* backing dev IO list */
58  #ifdef CONFIG_CGROUP_WRITEBACK
59          struct bdi_writeback    *i_wb;          /* the associated cgroup wb */
60
61          /* foreign inode detection, see wbc_detach_inode() */
62          int                     i_wb_frn_winner;
63          u16                     i_wb_frn_avg_time;
64          u16                     i_wb_frn_history;
65  #endif
66          struct list_head        i_lru;          /* inode LRU list */
67          struct list_head        i_sb_list;
68          struct list_head        i_wb_list;      /* backing dev writeback list */
69          union {
70                  struct hlist_head       i_dentry;
71                  struct rcu_head         i_rcu;
72          };
73          atomic64_t i_version;
74          atomic64_t i_sequence; /* see futex */
75          atomic_t   i_count;
```

```
76          atomic_t    i_dio_count;
77          atomic_t    i_writecount;
78  #if defined(CONFIG_IMA) || defined(CONFIG_FILE_LOCKING)
79          atomic_t                i_readcount; /* struct files open RO */
80  #endif
81          union {
82                  const struct file_operations    *i_fop; /* former ->i_op->
                        ↪ default_file_ops */
83                  void (*free_inode)(struct inode *);
84          };
85          struct file_lock_context *i_flctx;
86          struct address_space      i_data;
87          struct list_head          i_devices;
88          union {
89                  struct pipe_inode_info *i_pipe;
90                  struct cdev            *i_cdev;
91                  char                   *i_link;
92                  unsigned                i_dir_seq;
93          };
94
95          __u32 i_generation;
96
97  #ifdef CONFIG_FSNOTIFY
98          __u32                   i_fsnotify_mask; /* all events this inode cares about
                ↪  */
99          struct fsnotify_mark_connector __rcu    *i_fsnotify_marks;
100 #endif
101
102 #ifdef CONFIG_FS_ENCRYPTION
103         struct fscrypt_info     *i_crypt_info;
104 #endif
105
106 #ifdef CONFIG_FS_VERITY
107         struct fsverity_info    *i_verity_info;
108 #endif
109
110         void *i_private; /* fs or device private pointer */
111 } __randomize_layout;
```

50004 - Operating Systems - Lecture 15

Oliver Killane

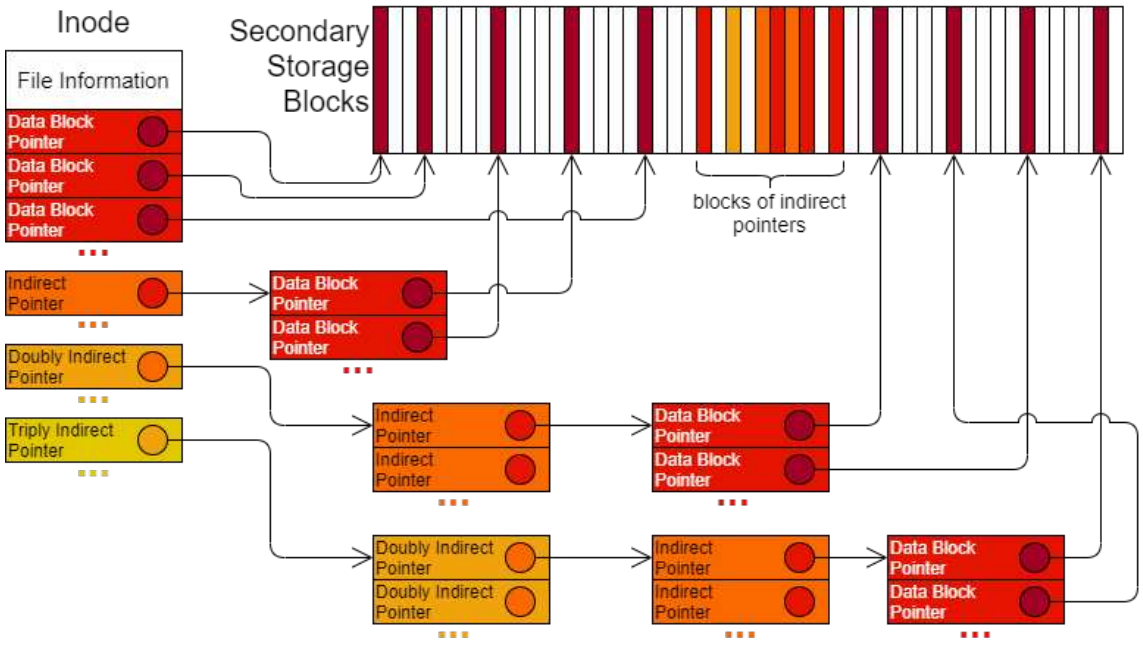
20/12/21

Lecture Recording

Lecture recording is available here

Linux Inodes Continued...

Indirect pointers are used to increase the number of data blocks that can be associated with a file. Indirect pointers can point to blocks, filled with either more indirect pointers or pointers to data blocks.



For example:
Take an OS with inodes containing:

- *6 direct pointers*
- *1 indirect pointer*
- *1 doubly indirect pointer*

Each pointer requires 8 *bytes, each block is* 1024 *bytes and each indirect block fills a whole block on the disk.*

What is the maximum file size?

Each indirect block can hold $\frac{1024}{8} = 128$ pointers to other blocks. Hence we can calculate the maximum number of blocks:

$$6 + 128 \times 1 + 128 \times 128 \times 1 = 16,518 \; blocks = 16,914,432 \; bytes \approx 16.13 \; MB$$

By adding a single triply indirect pointer we can increase this:

$$6 + 128 + 128^2 + 128^3 = 2113670\ blocks = 2,164,398,080\ bytes \approx 2.02\ GB$$

We can also determine the number of disk reads required for accessing a block.
Given an OS using inodes with 6 direct pointers, 1 indirect and 1 doubly indirect, with pointers being 4 bytes long and blocks 1024 bytes large.

Calculate the disk block reads required to get the 1020^{th} *data byte, and then* $510,100^{th}$ *data byte, assuming nothing is cached in memory.*

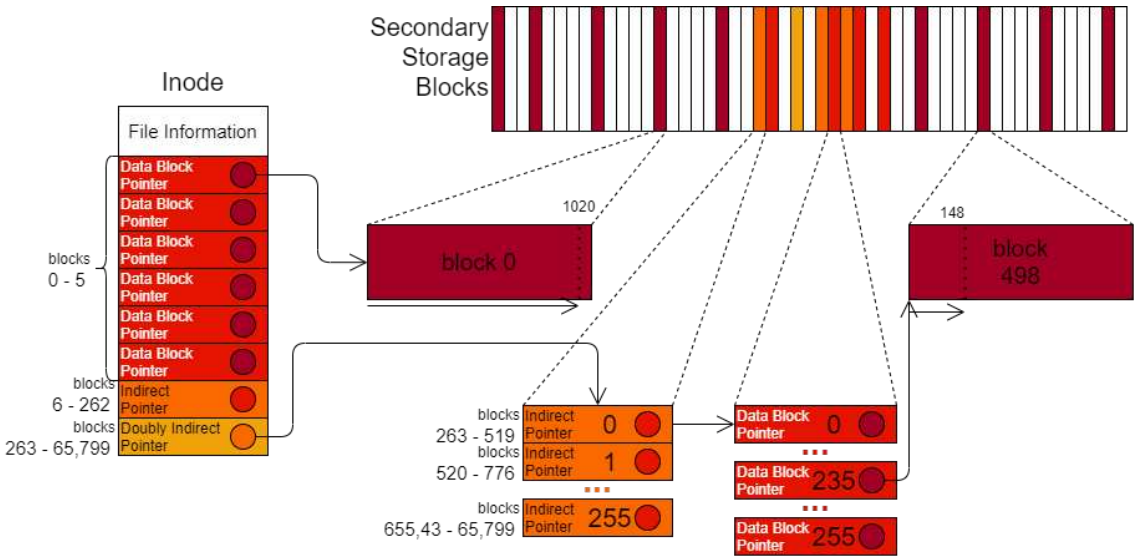
$1020^{th} byte$

$\lceil \frac{1020}{1024} \rceil = 1 \rightarrow 1^{st}\ block$. Hence 1 read required to get the inode, and 1 required to get the data from the block pointer & the offset of 1020 bytes.
Requires 2 *reads*

$510,100^{th} byte$

$\lceil \frac{510,100}{1024} \rceil = 499 \rightarrow 499^{th}\ block$. Each block of pointers contains $\frac{1024}{4} = 256\ pointers$. Hence this block is pointed to indirectly by the doubly indirect pointer.

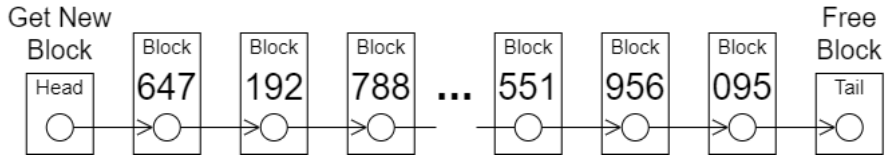
Therefore we require 1 read to get the inode, then 2 reads to get the data block, then 1 more to read from the data block.
Requires 4 *reads*



Free Space Management

When allocating a new block for a file, we need to quickly determine which is available.

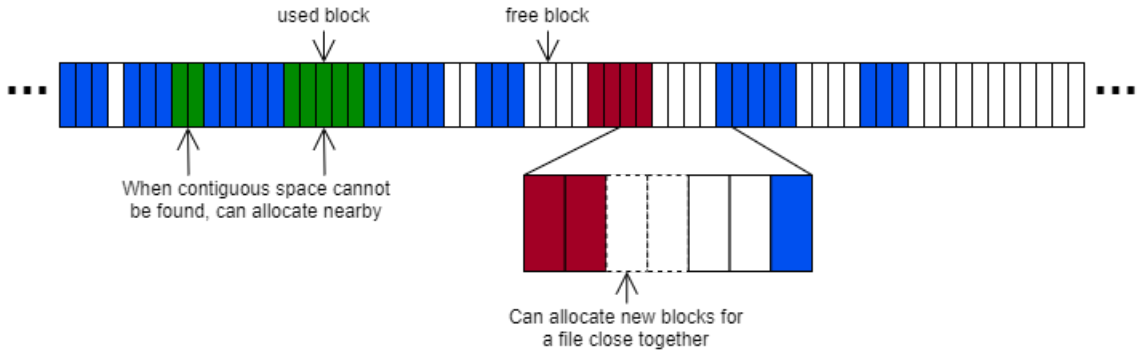
Free List



Linked list of blocks that are free. To allocate, take free blocks from the start of the list. To free, place block at the end of the list.

- Freeing and allocating blocks is fast($O(1)$), simply go to head/tail of the list.
- Files unlikely to be contiguously allocated. The location of an allocated block is effectively random, so when reading files, we must seek across many random locations on the disk which is slow.

Bitmap



Each bit represents a block at the same index.

- Uses little memory (single bit per entry).
- Can quickly determine contiguous blocks at locations (useful for placing file's blocks next to each other to reduce seek time when accessing).
- Highly optimised bit operations exist as instructions for most CPUs (allows for fast operations on the bitmap).
- May need to search the entire bitmap to find a free spot ($O(n)$ complexity) which is slow.

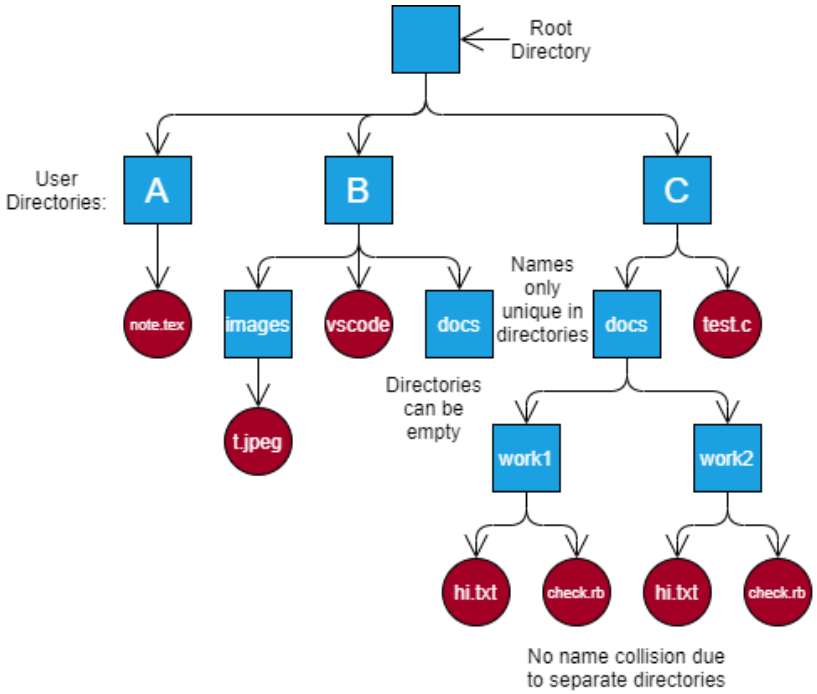
File System Layout

- **Boot Block** Used to store data to be loaded to start the OS.

- **Superblock** Contains file system information, including:
  - Number of inodes.
  - Number of data blocks.
  - Start of Inode & free space bitmaps.
  - Location of first data block.
  - Block size.
  - Maximum file size.

Filesystem Directory Organisation

- Map symbolic names (e.g "Lecture15.tex") to logical disk locations (e.g Disk 0, block 2354 (Logical Block Addressing)).
- Helps with file organisation.
- Prevents naming collisions (e.g names only unique inside a directory).



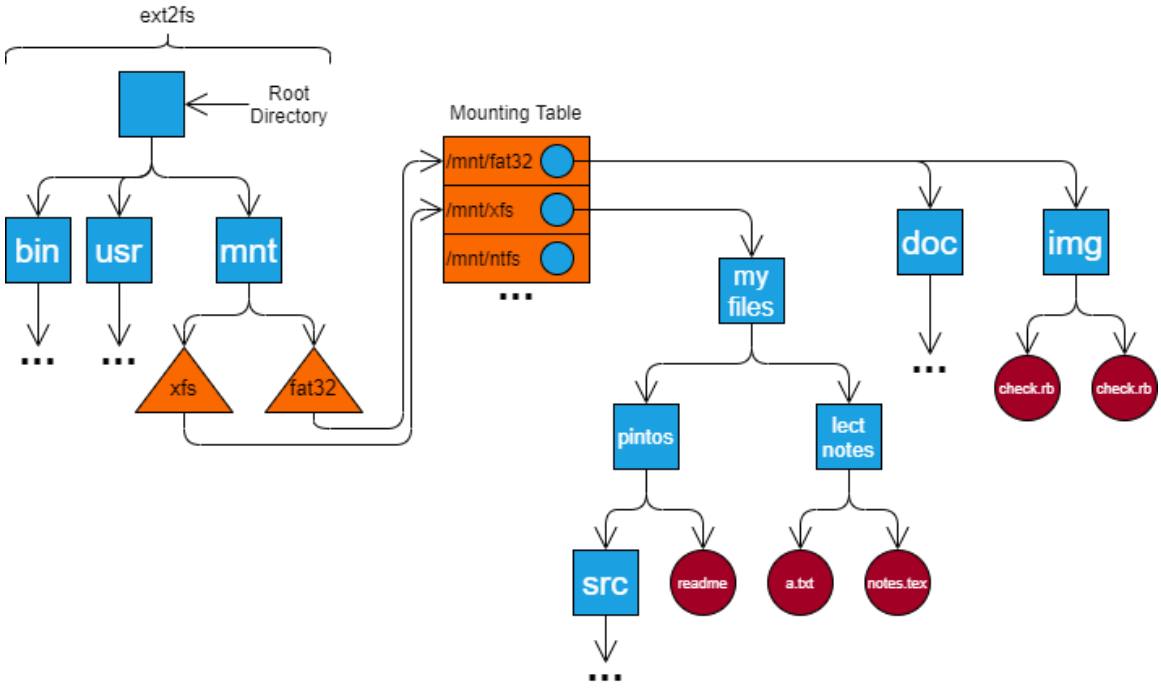
- **Pathnames**
  A path from the root directory to a file. Can be absolute (from root directory) or relative (from current/working directory). Delimited by "\" on Windows, and "/" on Linux/Unix.
- **Search**
  Can match pathnames with wildcards (e.g "*.pdf" (match all pdfs in current directory) or "lecture[0-9]/images" match all image directories from single digit lecture folders.)
- **Mount**
  Create a link to another file system such as a remote server or another disk (e.g usb drive).

  When mounting:

- Combine multiple filesystems into a single namespace (so can traverse through multiple).
- Allow reference from a single root directory (filesystem accessed as a directory from the root).
- Mount point is the folder from which the mounted filesystem can be accessed from the native one.
- Support soft links (not hard links as they are dependent of filesystem implementation).

This is managed using a mounting table, which records location of mount points and devices.

When a mount point is encountered the native filesystem consults the mounting table to determine which file system to go to.



- **Link** A reference to a directory/file in another part of the filesystem.
  - **Hard Link** Reference the address of the file (only allowed for files, not directories, in unix). A counter is used to check for when a file should be deleted (when no links).
  - **Soft/Symbolic Link** Reference the address using a directory entry (when deleting, leave links, raise an exception when the link is used).

  Must keep track of links to prevent looping when traversing the file system as a tree.

  Note that hard links are not allowed for directories as they may result in an island (where a hard link points to itself, or two mutually point to eachother) meaning they can never be deleted.

Unix/Linux Directory System

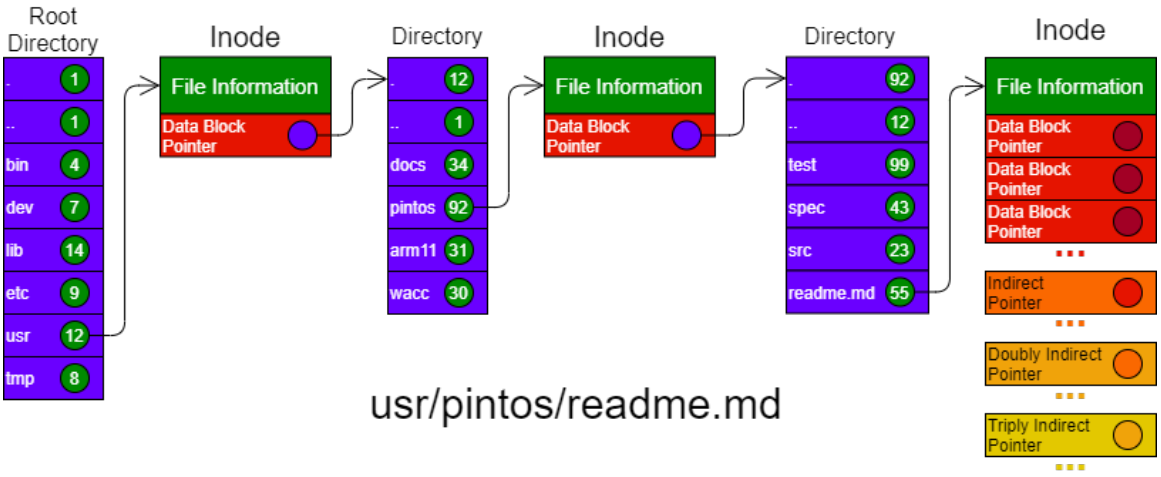
System Calls

```c
/* Create a new directory. */
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);

#include <fcntl.h> /* Definition of AT_* constants. */
#include <sys/stat.h>
int mkdirat(int dirfd, const char *pathname, mode_t mode);

/* Remove a directory (must be empty already). */
#include <unistd.h>
int rmdir(const char *pathname);

/* Create a new hard link to an existing file. */
#include <unistd.h>
int link(const char *oldpath, const char *newpath);

#include <fcntl.h> /* Definition of AT_* constants. */
#include <unistd.h>
int linkat(int olddirfd, const char *oldpath, int newdirfd, const char *newpath, int
    ↪ flags);

/* Remove a filename from the file system, if this is the last link, remove the
 * file.
 */
#include <unistd.h>
int unlink(const char *pathname);

#include <fcntl.h> /* Definition of AT_* constants. */
#include <unistd.h>
int unlinkat(int dirfd, const char *pathname, int flags);

/* Change working directory. */
#include <unistd.h>
int chdir(const char *path); /* Change using path. */
int fchdir(int fd);          /* Change based on file descriptor. */

/* Open a directory. */
#include <sys/types.h>
#include <dirent.h>

struct DIR {
    struct dirent ent;
    struct _WDIR *wdirp;
};

struct DIR *opendir(const char *name);
struct DIR *fdopendir(int fd);

/* Close a directory. */
#include <sys/types.h>
#include <dirent.h>

int closedir(struct DIR *dirp);

/* Read a directory. */
```

```c
#include <dirent.h>

/* Directory entry structure.  */
struct dirent {
    ino_t          d_ino;       /* Inode number. */
    off_t          d_off;       /* Offset to the next dirent. */
    unsigned short d_reclen;    /* Length of this record. */
    unsigned char  d_type;      /* Type of file; not supported by all file system
        ↪ types. */
    char           d_name[256]; /* Null-terminated filename. */
};

struct dirent *readdir(struct DIR *dirp);

/* Reset the position of directory stream DIRP to the beginning of the directory. */
#include <sys/types.h>
#include <dirent.h>

void rewinddir(struct DIR *dirp);
```

Directory Representation

Directories map symbolic names to inodes, these inodes can be other directories (typically only a single block) or files.



50004 - Operating Systems - Lecture 16

Oliver Killane

24/12/21

Lecture Recording

Lecture recording is available here

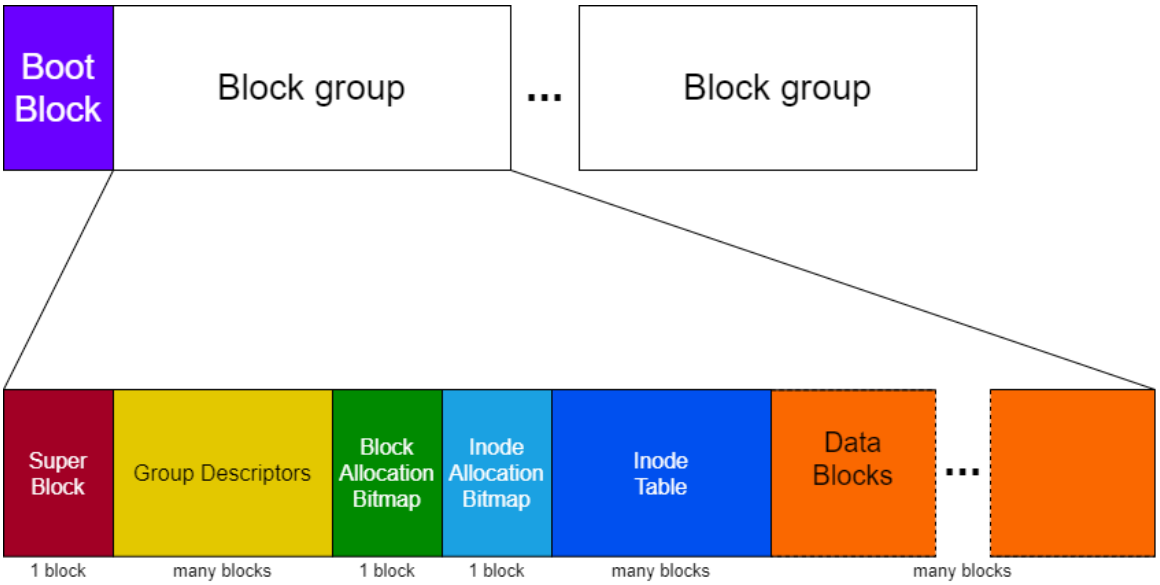
Linux ext2fs

The second **EXT**ended **F**ile **S**ystem is a high performance, robust file system (formerly the standard file system, now replaced by others but remains very portable).

- Typically uses block sizes of $1024, 2048, 4096, 8192$.
- 5% of blocks are reserved for the root (safety mechanism to ensure root processes can run even after a malicious process uses all available disk space).
- **ext2 inode** is used to represent files & directories. It stores access, permissions, and other metadata.
- **ext2 inode** uses 12 direct pointers, the 13^{th} is indirect, the 14^{th} doubly indirect and the 15^{th} triply indirect (allows fast access to small files, but allows for large files).

Block Groups

This filesystem makes use of **block groups** (clusters of contiguous blocks), and attempts to store related blocks (e.g from the same file) in the same block group to reduce seek time.



- **Super block** Critical information about the entire file system.
  - total number of blocks.
  - size of blocks, inode & block groups.
  - Metatdata for time accessed, mountings.

  Redundant copies are held in some block groups (reduces time to seek critical data).
- **Group Description** Contains pointers to the blocks containing:

  - Inode Allocation Bitmap
  - Block Allocation Bitmap
  - Inode Table
  - Other metatdata/accounting information.
- **Block Allocation Bitmap** Bitmap of blocks used/free within the data blocks section.
- **Inode Allocation Bitmap** Bitmap of entries containing an inode within the inode table.
- **Inode Table** Contains entries for every inode block in the group, spans over many blocks (many inodes, inode does not use up an entire block).
- **Data Blocks** Contains data blocks for files & directories.

In more detail ...

This excellent webpage goes into brilliant detail with example programs for **ext2fs**.

Security

Seurity Goals

The main objectives of security are:

- **Data Confidentiality** Preventing unauthorised read of data. e.g application secrets
- **Data Integrity** Preventing unauthorised write of data. e.g tampering with system clocks
- **System Availability** Preventing denial of service attacks. e.g monopolising a device (e.g hard drive)

Types of security in which we attempt to achieve said goals include:

- **People Security** Insider, social engineering (e.g blagging).
- **Hardware Security** e.g physical secureness of a system's hardware (e.g physically steal a disk to access data).
- **Software Security** Using flaws in software to access a system (e.g exploit a bug to run programs with higher permissions, such as superuser, without properly acquiring said permissions).

People Security

- **By Insiders** Need elevated privileges for their jobs, abuse these to compromise the security of the system.
- **Social Engineering** As people are often not security concious.

Phishing	Fake communications (e.g texts, emails) impersonating an organisation (can carry links to fake websites, or malicious attachments).
Blagging	More personal phishing (e.g ringing customer support).
Shouldering	Watching details (e.g passwords) in person.

  And many more.
- **Convenience** People ignore security requirements/sidestep them as it is easier (e.g repeated use of passwords, not properly verifying emails).
- **Ignorance** Lack of knowledge on security & incorrect assumptions (e.g email sender cannot be forged, internal emails are safe etc).

Hardware Security

- **Physical Access** Once physical access is acquired by the malicious party, they could:
  - Read & alter contents of attacked disks, memory.
  - Snoop on and forge network traffic (e.g if the machine is whitelisted for an internal organisation network).
  - Damage the machine to remove functionality (e.g commit arson against computer).
- **Hardware Flaws** Such as side channel attacks, badly implemented access control (Meltdown).

Side Channel Attacks

Gaining information using the implementation of a computer system. For example:

- **Cache Attack** Using cache accesses (e.g to a shared cache) by a victim to gain information.
- **Timing Attacks** Measuring time taken by a victim to gain information (e.g time taken to process a given key, compared with victim's unknown one).
- **Data Remnance** Reading data after the user has deleted it (e.g still in memory and has not be reset, files deleted on a drive are still present - blocks have just been marked as free).

Meltdown

Attack notes in **Lecture 10**. Modern CPUs reorder and speculatively execute instructions. By speculatively executing instructions that should fault (as they are accessing an invalid result) which change the state of cache based on the data accessed data can be improperly accessed.

1. CPU reorders instructions, speculatively executes access at **VICTIM_ADDR**.
2. If the data speculatively accessed is **0** then access **ASSAILANT_ADDR_F** else access **ASSAILANT_ADDR_T**.
3. The speculative executions is ignored (e.g if statement was **false**).
4. The assailant process now times its access to **ASSAILANT_ADDR_F** and **ASSAILANT_ADDR_T**. If the access is fast, the data is in cache, and hence we know the bit at **VICTIM_ADDR**.

Software Security

Compromising a system (unauthorized read, write & service denial) through software.

Common methods for exploits include:

- **Buffer Overflows** For example using **strcpy** on strings. If the string is larger than the buffer it may overflow, overwriting important program data. Use **strncpy** instead to limit bytes copied to the size of the buffer.
- **Integer Overflows** An example of this was a flaw in **SSH** you can read about here. By using integer overlow, **SSH** can accidentally make a hash table of size zero, effectively resulting in a buffer overflow when attempting to add data to the hashtable.

- **String Formatting** For example the **Log4J** vulnerability originating from string formatting in log messages allowing use of the **JNDI** java feature (which gets and locally executes java code). There is a great video explaining it here.

Access Control

- **Principals** Users, User groups or processes that may want to perform actions.
- **Authentication** Verifying the identity of users (**principals**). (e.g login system)
- **Authorisation** Only allow principals to perform actions when authorised. (e.g enforce read-/write/execute access on files)

Authentication

To authenticate the identity of a user we can use:

- **Personal Characteristics** Key based on user (usually biometrics).

  - Fingerprints.
  - Retinal scan.
  - Facial Recognition.
  - Signature & Signature motion analysis.
  - Typing rhythm analysis.

  Advantages include:

  - Characteristics hard to forge.
  - Convenience (e.g fingerprint and facial recognition becoming standard on smartphones).

  Disadvantages include:

  - Require special hardware (can be expensive).
  - Can have false positives/negatives (error associated with readers - e.g dirt on fingers, lighting conditions for facial recognition).
- **Possessions** Key is an item securely-kept by the user. This is the most widely used system.

  - RFID cards.
  - Implants.
  - Secure keys (e.g Yuibikey)
  - Can also consider devices such as smartphones, smartwatches which can be used as keys (e.g often for **2fa** - two factor authentication).

  Advantages include:

  - Convenience (rfid cards can be cheap to buy & replace).

  Disadvantages include:

  - If lost or stolen, can be used to impersonate.
  - Sometimes the cost of devices can be high (e.g complex locks, card scanners)
- **Knowledge** Key is a secret known to the user. (i.e passwords) Password turnover is important (longer a password is used, the greater time a cracker has to guess the password).

  A strong password (few english words, effectively random) can only be cracked by exhaustive search/brute force.

  Advantages include:

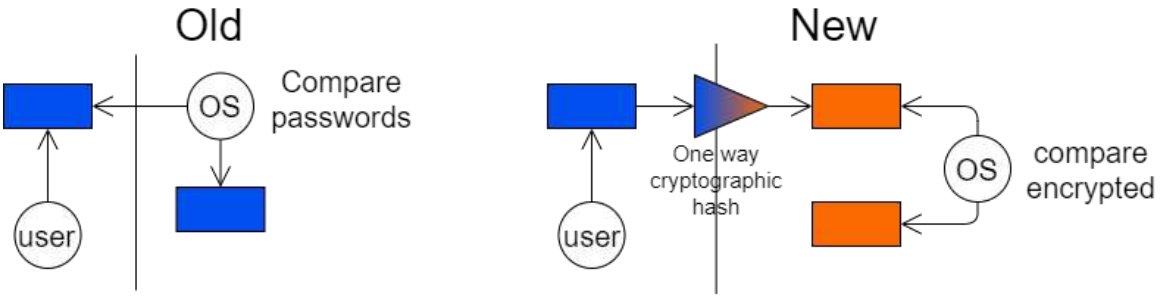
– Very cheap to implement (requires no special hardware).
– If password is kept secret, the system is secure.

Disadvantages include:

– Security diminished when passwords are reused.
– If passwords are stored, they are only as secure as the storage.
– Dictionary attacks be used for common passwords, or ones composed of words.
– If passwords are reused, the security of the systems using them are only as good as the weakest system using the password.

Password Verification

To verify passwords older systems store passwords in a protected file, newer systems store a one-way cryptographic hash of the password, and compare the hashed password entered with this. This ensures that even if the protected file is accessed, it is not possible to determine the password.



The older system is vulnerable to passwords being accidentally disclosed by a sysadmin, or if the sysadmin user is compromised.

The **etc/passwd** file used to contain plaintext passwords, currently holds user data.
The **etc/shadow** file contains hashed passwords.

One-Way Hash Function Password Protection

Can compute a hash easily, but not possible to invert the function.

$$Hash(Data) = Encrypted$$

The only feasable way of getting *Data* from *Encrypted* is brute force/extensive search of all data.

UNIX systems use hashing based on **DES** (Data Encryption Standard)

Rainbow Tables

Given a one-way hashing function, we can compute a table of hashes for all the most popular passwords.

This way given a hashed password, we can check if it is in the table. If it is, we know the password associated with the hash. We only need to compute the hash for each password once, and can continue to improve the table over time.

Password Salting

A method to prevent rainbow tables being effective. Another value s (the salt) is used.

The salt is created randomly upon account creation, and is stored with the hash (which takes the salt and password as parameters).

$$userid,\ salt,\ Hash(salt, password)$$

When the user logs in, the salt is used with the entered passsword to compute the hash.

$$\text{re-compute } Hash(salt \text{ (stored)}, password \text{ (entered)})$$

Hence computing a rainbow table for the hash in infeasible as for every password added to the table, every possible salt value must be hashed with it. This also ensures that two identical passwords are extremely unlikely to have the same hash.

Adobe Password Leak

In november 2013 $130,324,429$ passwords were leaked. As salting was not used, rainbow tables could be used to identify passwords.

Place	count	ciphertext	plaintext
1	1,911,938	EQ7fIpT7i/Q=	123456
2	446,162	j9p+HwtWWT86aMjgZFLzYg==	123456789
3	345,834	L8qbAD3jl3jioxG6CatHBw==	password
4	211,659	BB4e6X+b2xLioxG6CatHBw==	adobe123
5	201,580	j9p+HwtWWT/ioxG6CatHBw==	12345678
6	130,832	5djv7ZCI2ws=	qwerty
7	124,253	dQi0asWPYvQ=	1234567
8	113,884	7LqYzKVeq8I=	111111
9	83,411	PMDTbP0LZxu03SwrFUvYGA==	photoshop
10	82,694	e6MPXQ5G6a8=	123123

Authorisation

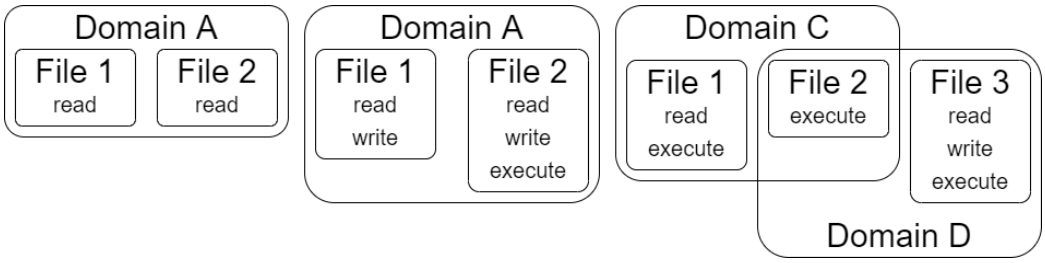
Determines who can access what objects (files, devices, directories, processes), and how they should access.

Principle of Least privilege

User should be given the minimum rights/privileges to carry out their assigned task.

This is not the case on many systems, to their detriment (as users can maliciously or accidentally misuse higher privileges).

Access Rights are defined as a set of objects, with operations permitted on them. Each **principle** has a **domain** specifying the access rights.



An **Access control matrix** specifies the rights of each principal over all objects. An actual matrix is not used in reality, due to its enormous size over thousands of principals and millions of objects.

	Object 0	Object 1	Object 2	Object 3	Object 4
Principal 1	read		read & write	execute	
Principal 2		read & write		read & execute	
Principal 3	read & write	read	execute		read
Principal 4		read & execute			read, write & execute

Access Control Matrix Implementation

As a 2d array would be two large, there are two main implementations:

- **Access Control Lists** Each column of the matrix is stored as a list, and associated with each object.

  This list enforces which principals can access the object, and what operations they can perform.

- **capability List** Rows of matrix as lists associated with principals.

File Access on UNIX/Linux

- Users are the principals, and each has an associated user ID (UID). Each user can belong to one or more groups.
- The superuser (**root**) has UID 0 and all access rights (can access any file with all operations).
- Files are objects (e.g files, directories, sockets, pipes, block and character devices). Each file can belong to at most one group.

Access operations are read, write and execute:

Access Right	**File**	**Directory**
Read	Can read file contents.	Can list directory contents.
Write	Can write to file contents.	Can create/delete owned files.
Execute	Can execute the file (create as a new process).	Can enter directory & get access to the files.

Displaying Permissions

When listing a directory with the **-l** flag, the file permissions, owner, and other metadat is displayed. This is in **Lecture 14** under **file attributes**.

When a file is executed, it executes with the privileges of the user executing. Unless the **SUID** is set. In this case the file runs with the owner's privileges when executed (for example temporarily increasing privileges, but only for that program).

As a result each user effectively has 3 user IDs:

1. **Real UID** ID of the user that started the process
2. **Effective UID** Effective ID of the process (used for access control checks), when executing a file with the **SUID** flag this will be different to the real **UID**
3. **Saved UID** UID the effective UID can be switched to.

- When executing a file, the effective UID is $\begin{matrix}\text{user's real UID if not SUID}\\ \text{file owner's UID if SUID}\end{matrix}$.
- A non-root/superuser process may temporarily reduce its privileges, saving its effective UID as the saved UID so it can be restored later. It may do this to ensure **principle of least privelege**, and only hold elevated access rights when they are necessary.
- A non-root/superuser process can set their effective UID to their real UID or saved.

50004 - Operating Systems - Lecture 17

Oliver Killane

27/12/21

Lecture Recording

Lecture recording is available here

File Access on UNIX/Linux continued...

Capability Lists

Effectively lists of the rows of an access matrix, associated with the principals (users). If a capability is in the list, then the user associated with it has the capability to perform said action on said object.

- Capabilities are protected pointers to objects, which specify the permitted operations (e.g file descriptor, index of capability list).
- Often implemented as protected pointers to objects. The kernel holds the capability information (can create, destroy and modify it), providing access to the user indirectly (e.g user has a file descriptor, or a key of sorts).
- Alternatively the capability may be held in user-memory, but encrypted.

Access Control Lists vs Capabilities

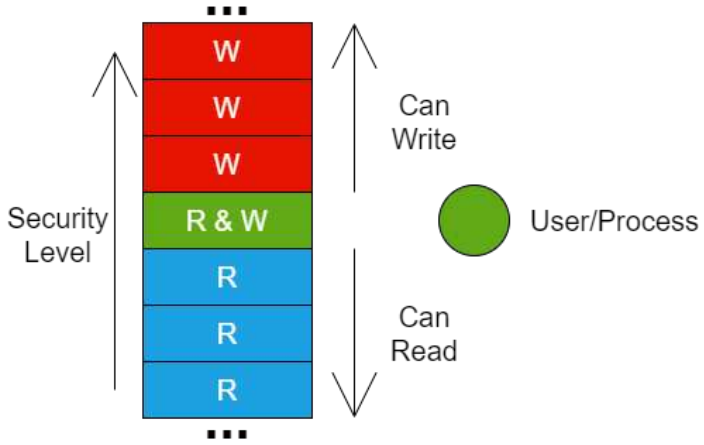
Action	**Best**	**Access Control Lists**	**Capabilities**
Principle of Least Privilege	Cap	Use of groups mean all users in a group get access rights.	Can be very fine grained, only giving capabilities to specific users.
Revocation	ACL	Can revoke all access by changing one ACL for the object.	Must search all capability lists, and remove capability.
Rights Transfer	Cap		Can transfer a single capability from one user to another.
Persistence	ACL	File systems are designed to be persistent.	Very complex.

DAC vs MAC

- **Discretionary Access Control (DAC)** Principals determine who may access their objects. This is the default in UNIX/Linux, file owners determine the access rights.
- **Mandatory Access Control (DAC)** Precise system rules determine object access. A policy determines access controls for files, devices etc.

Bell - La Padula Model

A **MAC** policy where information cannot travel down the security hierarchy.

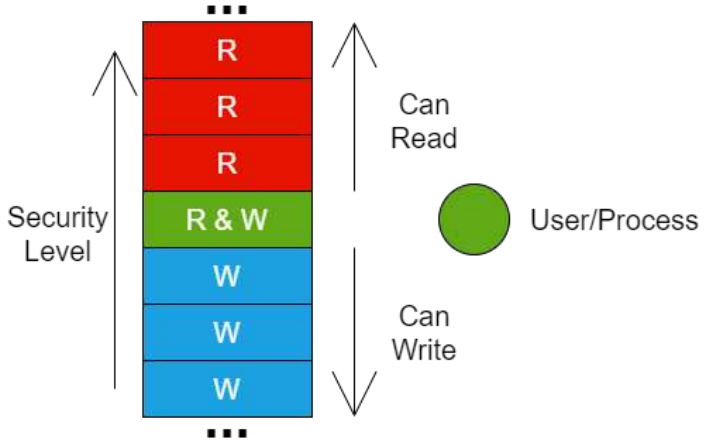


- Objects and principlals are assigned a security level (e.g unclassified, confidential, top secret)
- **Two Rules**
  - Simple Security Policy — Processes can only read objects at its security level or lower.
  - The * property — Processes can only write to objects at their security level or higher.

Information can only travel up the security levels. This ensures confidentiality, but not integrity.

Biba Model

A **MAC** policy guaranteeing data integrity.



- **Simple Integrity Principle** Can only write to objects of the same or lower security level.
- **Integrity * Property** Can only read objects of the same or higher security level.

Design Principles

- **Least Privilege** Each process should run with the lowest privilege possible. Default should be no access.

- **Simple & Uniform Mechanism** ensures design is easy to reason about, and can be applied to many types of principals & objects.
- **Psychologically Acceptable** If policies are too much of a burden, too complex, or difficult to deal with then people will not implement them.
- **Public** Security by obscurity is bad, by making the design public it can be critiqued for flaws.

Virtualisation

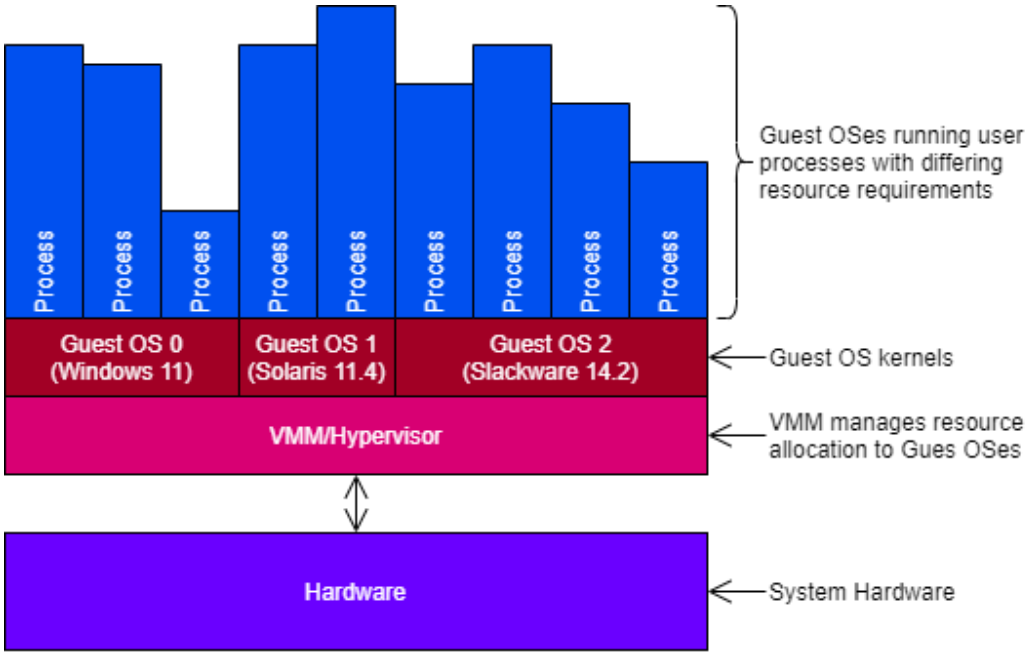
Operating systems abstract away hardware, providing abstractions such as virtual memory, signals, file systems, syscalls and more.

For some systems we may want to abstract hardware, providing virtualised CPUs, disks, interrupts, physical memory etc as a virtual machine. such that an entire operating system can run inside.

This is useful for:

- **Legacy software** Can run software designed for much older systems.
- **Security** Even if virtual machine compromised, the system has not. This is especially useful when deliberately compromising a machine to reverse engineer malware, virtual machine acts as a secure sandbox in which to experiment.
- **Software Management** avoids issues with conflicting software (versions and dependencies) by running them in separate virtual environments.

Top achieve this we use a **Virtual Machine Monitor** (**VMM**) or **Hypervisor** to partition resources, and provide these abstractions as a virtual machine on which guest Operating Systems can run.



50004 - Operating Systems - Lecture 18

Oliver Killane

28/12/21

Lecture Recording

Lecture recording is available here

Benefit of Virtualization

- **Consolidation** RUnning many servers on one.
  An organisation may want to run many dedicated servers (e.g for mail, webserver). However each will not use an entire machine's resources all the time.

  Hence can save money by running multiple dedicated servers on the same physical server, dynamically allocating resources on demand.

- **Legacy Software** Some software is OS & version specific.
  Despite large efforts by companies to ensure backward compatability, software produced for older OSes may be incompatible/buggy on newer ones.

  Applications may use drivers that are no longer supported for legacy hardware that is no longer commercially available.

  e.g Windows 8, 7, Vista are not truly backwards compatible with XP

- **Software Development & Testing** Test more & more easily
  - Can test multiple instances of an Operating System simultaneously on a machine.
  - Can test system crashing errors (e.g kernel panics) without brining the machine down.

- **Security** Can isolate virtualized system.
  - Only the **VMM** runs in kernel mode, hence if a guest OS is compromised, the whole system & other guests are not.
  - **VMM**s are typically significantly smaller than OS kernels. Less code → less potential for exploitable bugs.
  - Intrusion detection via introspection on guest OSes (monitor resources, e.g scan memory) for suspicious activity.

- **Software Management** Simpler system administration.
  - Can take snapshots of the state (memory, CPU, etc) of a guest OS, and roll back to snapshots.
  - Can migrate between hosts (switch guest OS between machines, take snapshot then transfer and resume). We can use this for load balancing (move to higher spec machine when it is required).
  - Applications depend on OS, compiler, library versions which sometimes conflict between applications. when we want to run a specific application, we can create a VM with all its requirements and then distribute this.

Implementing a Virtual Machine Monitor (VMM)

Technique	Description	Slowdown	
		CPU-Bound	IO-Bound
Emulation	Intercept all instructions. The emulate their execution on hardware.	$\approx 100\times$	$\approx 2\times$
Modern VMs	Use hardware to accelerate virtualization.	$\approx 5\%$	$\approx 30\%$

Instructions

Most instructions can be run directly on the CPU. We may need to perform some checks (e.g to not access other guests' memory).

```
1  ; MASM (Microsoft macro ASseMbler) syntax for convenient lstlisting highlighting
2  mov eax, [ebp + 16]          ; eax =  *(ebp + 16)
3  lea eax, [ebx + 8]           ; eax = ebx + 8
4  call printf                  ; call to printf
5  add eax, DWORD PTR 12[ebp]   ; eax += 4 bytes below
6  inc eax                      ; eax++
```

Only sensitive instructions need to be intercepted & emulated. For example with:

```
1  cli ; Clear Interrupt Flag (maskable interrupts are ignores - e.g timer)
2  sti ; Set Interrupt Flag (enables interrupts after next instruction)
3
4  ;When recived, the VMM uses a trap:
5  ;VM #3:
6  ;1. cli
7  ; - [trap, cli in user mode!] VMM stops delivering interrupts to VM #3
8  ; - VMM remembers IF (interrupt flag) state
9  ; - VM#3 continues
10 ;2. interrupts disabled
11 ;3. ...
```

When the instruction is run, it traps, allowing the **VMM** to take over and run relevant routines on the guest OS that caused the trap.

However there are some obstacles:

- **Some instrtuctions do not trap**
  For example on the intel i386 **popf** can be used to replace flags. Including the **IF** (Interrupt Flag) bit.

  However if running in user mode, the **IF** flag is left unchanged, with no trap.

  Hence as the guest OS runs in user space, this instruction could silently fail (not informing VM of attempt to access **IF**)

- **Visibility of privilege level**
  The **CS** (code segment) register informs a process of its privilege level. If we do not attempt to hide this from the guest OS, it will be able to determine if it is in a virtual machine as when running in a vm it will not have kernel privilege on the hardware.

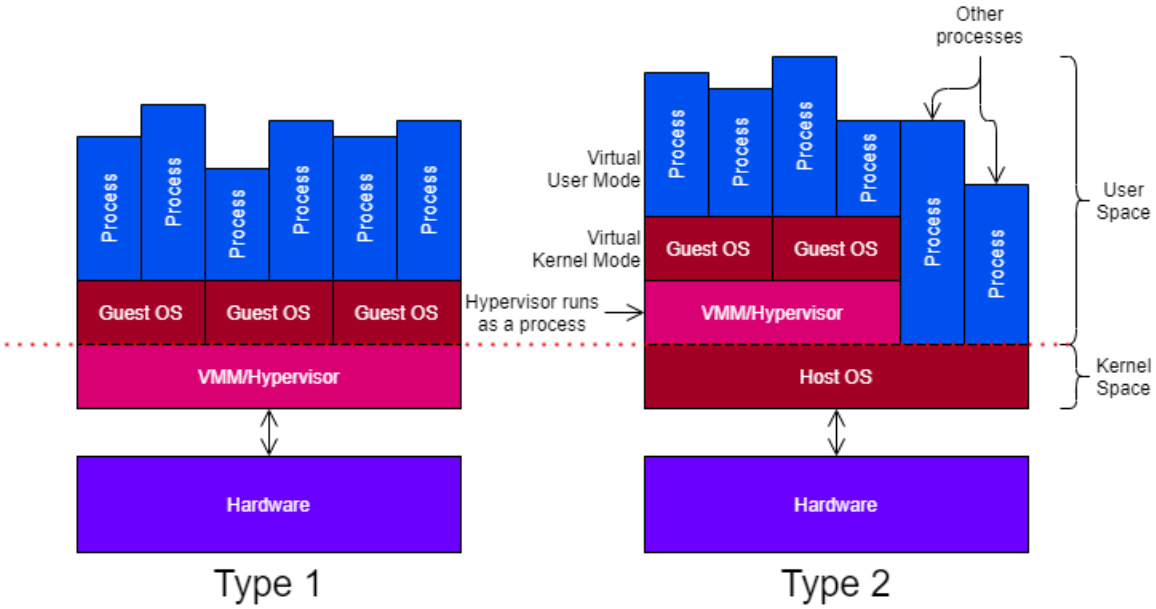
A CPU is considered **virtualizable** if all sensitive instructions trap, and hence when a sensitive instruction is executed the **VMM** can take over.

x86 has been **virtualizable** since 2005:

- Intel Virtualization Technology **VT-x**
- AMD Virtualization **AMD-V**
- And many more extensions such as **VT-d** and **APIC-v** (**APIC** - Advanced Programmable Interrupt Controller)

Hypervisors

Hypervisor Types



- **Type 1** Runs on bare metal - Better performance.
  Has access directly to hardware and runs in kernel mode.

  Requires hardware to support virtualization (trap privileged/sensitive instruction)

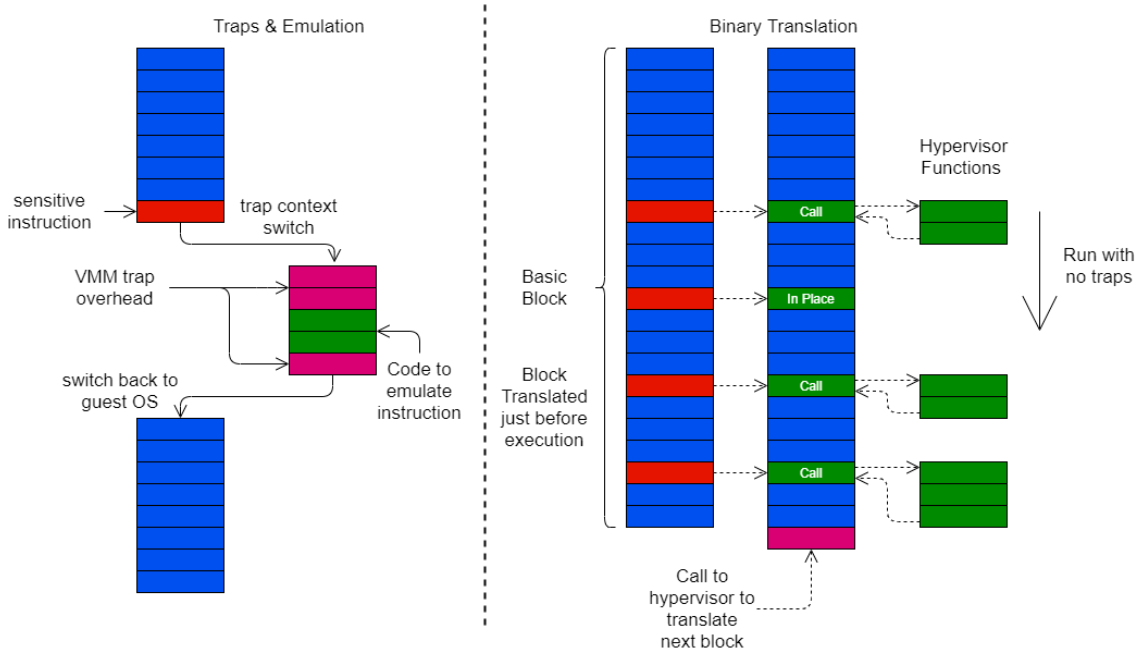
- **Type 2** Runs inside host OS Hypervisor is a program that can be run as a process.
  - Any performance or security issues in the host OS can affect the guest OSes.
  - Easy to install (like a regular application).
  - Can use drivers, scheduling and other services from host OS.

Most systems use a hybrid of both approaches.

Binary Translation

Frequent traps incur significant performance overhead (must clear **TLB**, other caches get obliterated, reduces effectiveness of branch prediction).

In order to eliminate traps, we can instead dynamically translate trap-inducing instructions into the code to resolve them. This is faster than trapping, then emulating, and then switching back to the guest OS.



This technique was originally developed for the **Type 2** hypervisor VMWare Workstation (though has been used outside of virtualization for decades, first paper I could find here.) It can also be used to speed up **Type 1** hypervisors.

1. Scan each **Basic Block** just before execution.
2. If the block contains sensitive/privileged instructions, replace them with calls to hypervisor functions, or instructions in place.
3. Replace last instruction with a call to the hypervisor (to translate next block of instructions and then resume there).

There are several possible optimisations:

- Cache translated blocks (do not need to redo translation when executing code again).
- Once the successor basic block has been translated, replace the hypervisor call with a normal jump.
- Do not translate user code (no privileged instructions so no need to translate, though may need address translation)

With these optimisations we must be careful for cases such as evicting cached blocks (may need to undo some hypervisor call $\rightarrow$ jump optimisations).

This can also slow down non-privileged instructions. For example user-level calls now need to be handled by the **VMM** to ensure only translated blocks are executed, and the **VMM** must be careful with saving pointers (e.g in call to **VMM** function) to ensure the guest OS cannot discover they are being virtualized through stack inspection.

Paravirtualization

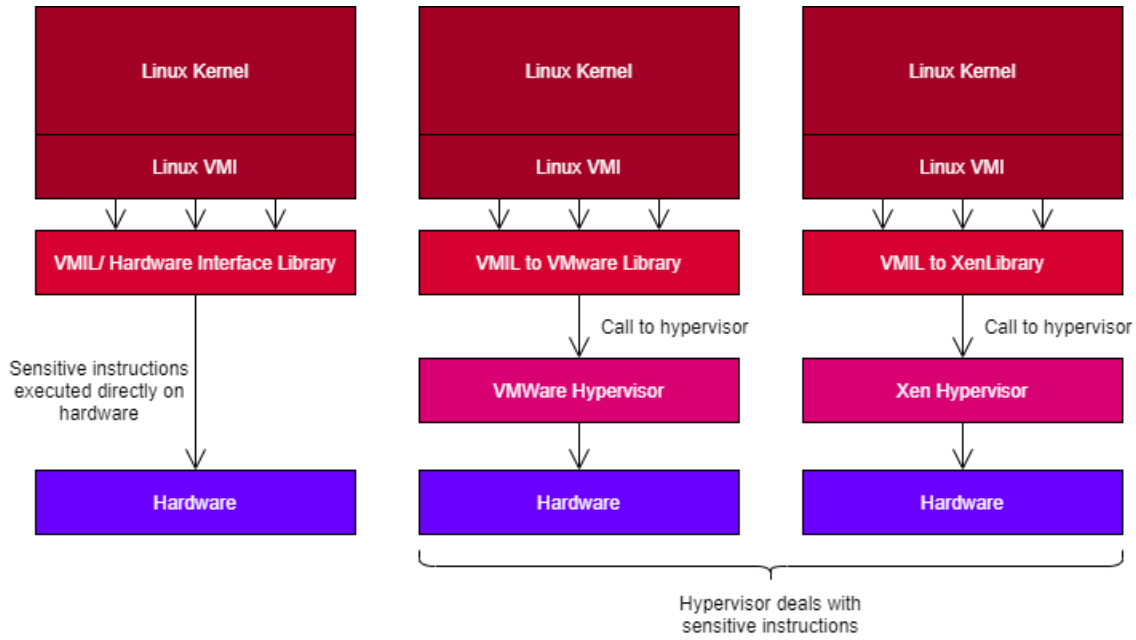
Another technique is to change the guest operating system's kernel source code to replace sensitive instructions with hypervisor calls.

- Can handle unvirtualizable hardware.
- Can achieve better performance.
- Need to make many changes to the guest OS source (not always possible). An OS must support it, as well as native hardware, as well as other hypervisors.

Virtual Machine Interface (VMI)

A single hypervisor **API** that can interface with multiple hypervisors & hardware. This **API** can connect with hypervisor specific libraries, or directly to hardware.

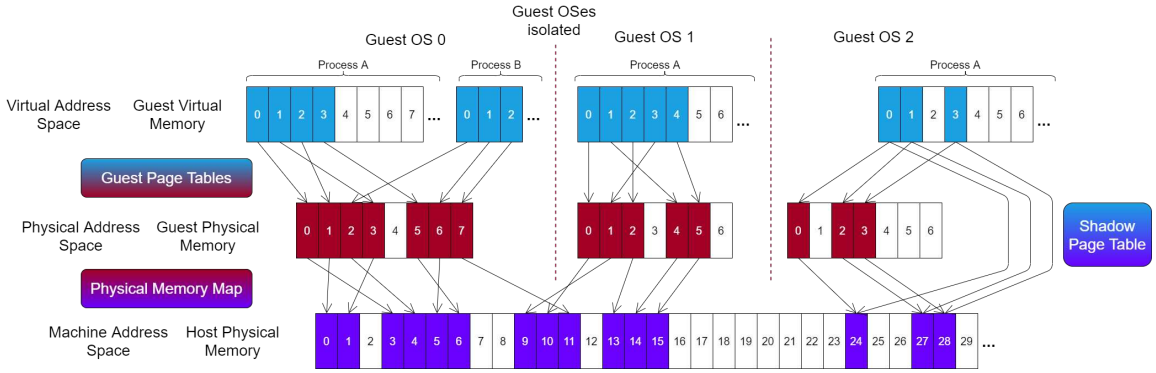
This can be achieved through function pointers. sensitive instructions run through function pointers, if virtualized, the pointer goes to a hypervisor specific function to resolve sensitive operation.



Memory Virtualization

We must prevent conflicts between guest OSes (where two guest OSes use the same frame in the host's physical memory). Hence we add an extra layer of indirection to translate:

Guest Virtual Addresses → **Guest Physical Addresses** → **Host Physical Addresses**



- Virtual Address Space uses Virtual Page Numbers (VPNs)
- Guest Page Tables translate VPNs → PPNs
- Physical Address Space uses Physical Page Numbers (PPNs)
- Physical Memory Map (**pmap**) translates PPNs → MPNs
- Machine Address Space uses Machine Page Numbers (MPNs)

Shadow Page Tables

Shadow Page Tables combine the two stage process of translating through the guest page tables and the physical memory map into one. The Guest OS is unaware of the shadow page table (managed by the hypervisor), which contains direct mappings to speed up address translation. When updating the shadow page table, the guest page tables & physical memory map are used.

- Hardware **MMU** (Memory Management Unit) uses shadow Page Tables.
- Hardware **TLB** maps VPNs → MPNs

On a **TLB** miss:

1. **Search Shadow Page Table**
   If the mapping found, update TLB & restart the instruction.
2. **Page Fault Handled by the VMM**
   (a) VMM attempts to find the **VPN** → **PPN** mapping in the guest OS page table.
   (b) If not there, then it is a **true page fault** to be handled by the Guest OS.
   (c) If it is found, it is a **hidden page fault** (Guest OS unaware). **VMM** updates the **pmap** and the shadow page table.

In order to use **Shadow Page Tables** they must be kept in sync with the Guest OS's Page Tables. This can be achieved in two ways:

Software

Mark page tables as read only, this way when the Guest OS writes to a page table, it traps and the **VMM** can take over to update both it & the Shadow Page table.

Traps are costly!

Hardware

Newer CPUs have hardware Support. The hardware itself does the required lookup on TLB miss, removing the need to use a shadow Page Table. (hardware implements the behaviour of a shadow page table).

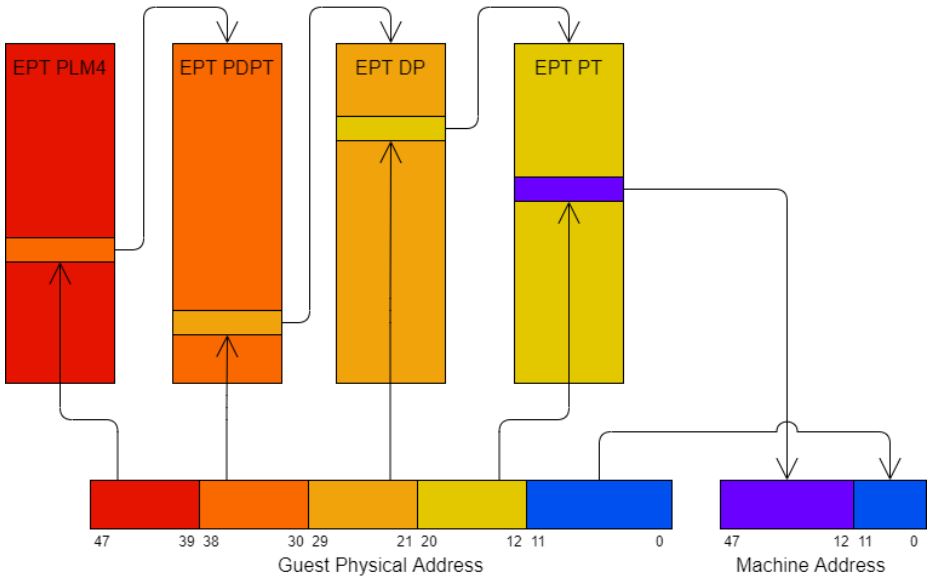
- AMD's Nested Page Tables (NPTs)
- Intel's extended Page Tables (EPTs)

On a **TLB** miss:

1. **MMU** searches for mapping in guest page table.
2. If not mapping found, it is a **true page fault**, hand control back to guest OS to resolve.
3. If mapping found:
   (a) MMU searches for mapping in **pmap**
   (b) If mapping found, update TLB & restart instruction.
   (c) else it is a **hidden page fault**, hand control to **VMM** to resolve it.

Intel EPT

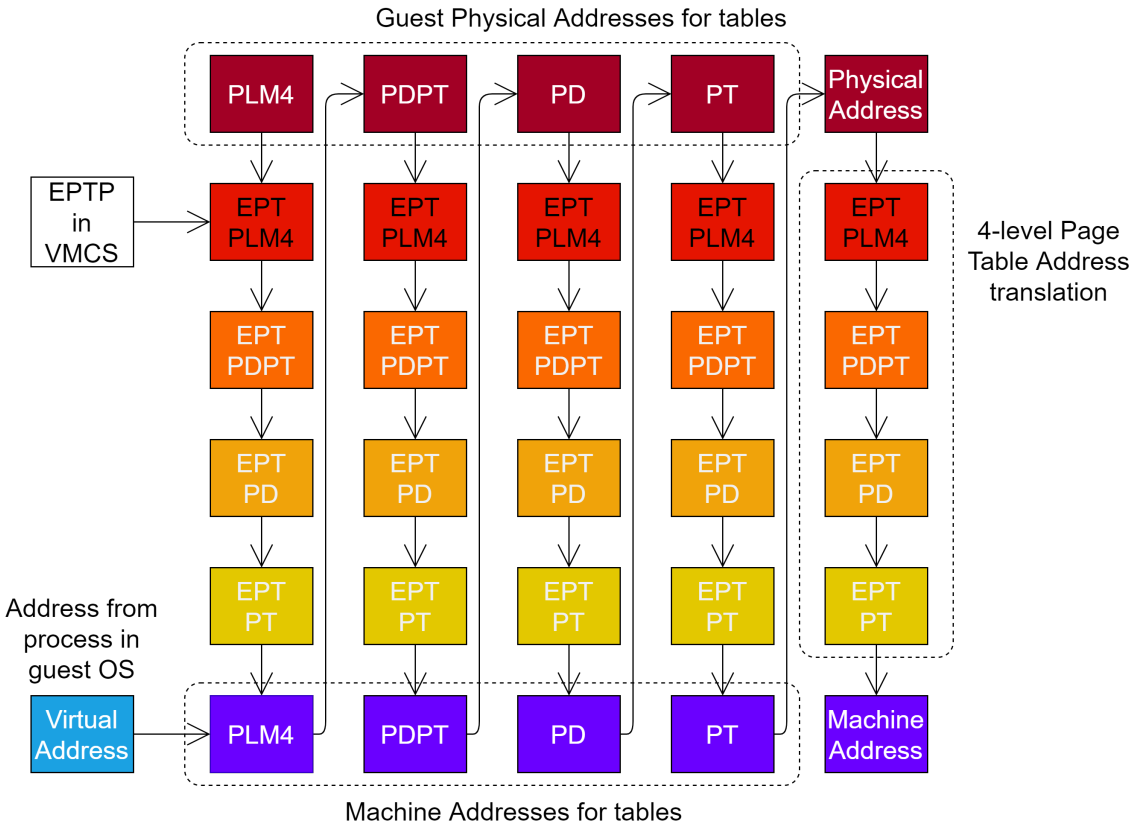
Intel Extended Page Table make use of a 4-Level page table as below:



In a **TLB** cache miss on a physical address (PPN) (e.g the hypervisor's page table), only 4 memory

accesses are required (4 level page table).

However when translating from virtual addresses (from the guest OS) far more memory accesses are required as translations are required for the 4 tables (24 accesses).



Paging Problems

The combined guest OS's physical memory can be larger than the Host's physical memory. In order to ensure the physical memory of Guest OSes is usable, we must swap guest physical pages to and from host physical memory.

- Hypervisor has little information on which pages the OS is using, Guest OS knows more about active pages.
- **Double Paging Problem**

  1. **VMM** select page P to be evicted/paged out.
  2. Guest OS selects page P to be evicted/paged out, accesses P brining it back from swap.
  3. Guest OS writes P to virtual disk.

  Here we have brought back a page, only for it to be immediately written to virtual disk.

50004 - Operating Systems - Lecture 19

Oliver Killane

30/12/21

Lecture Recording

Lecture recording is available here

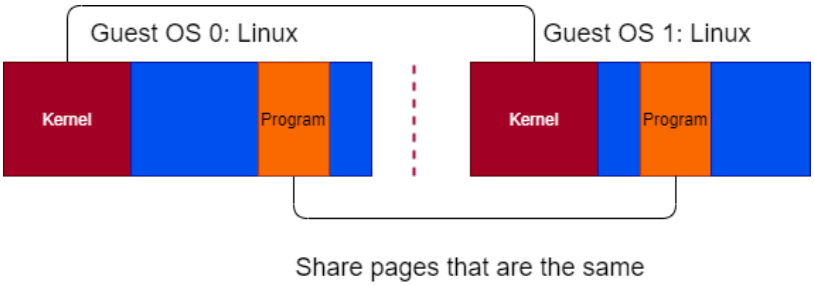
Memory Virtualization Continued...

Balloon Driver

Instead of relying on the **VMM** to determine pages to evict, force the Guest OS to evict pages.

- **Balloon Driver** installed in OS.
- Memory allocated to the balloon is available to the **VMM**.
- When the balloon is *inflated*, memory is reclaimed by **VMM** as low memory prompts the Guest OS to evict pages.
- When balloon is *deflated*, more memory is available to Guest OS.

Sharing Memory between VMs



By sharing identical pages (e.g for kernel, program information when VMs are on the same OS), memory usage is reduced.

Comparing pairs of pages would be too expensive, instead contents of pages is hashed. Only pages with identical hashes are compared, and if identical, shared.

There are many possible policies for when, and how to determine which pages to share. For example VMWare's ESX attempts to share pages when paging out to disk, and scans guest pages randomly at a predefined rate.

Containers

Processes are already isolated in terms of:

- Non-privileged CPU instructions & registers
- Virtual Memory
- File System
- System Calls (e.g for I/O)
- Signals

To improve this isolation between processes:

- **Support for Namespaces** e.g process identifiers are allocated and unique in groups, rather than process identifiers being global.
- **Separate Root Filesystems** Split the filesystem into many smaller, separate filesystems. Processes can create their own **jails** to ensure file system isolation.
  Useful for security (e.g split a companies internal and external services through jails), with very little overhead.

This is the main motivation behind containers.