Equation Sheet (Y3S1) – Concurrent Systems

1. Syntax & Concurrency Basics

(none)

2. Deadlock

(none)

3. Operating Systems

$$t_t = t_{comp} + t_{data} + t_{overhead} \text{ (total)}$$

$$t_{overhead} = t_{polling} \text{ (polling)}$$

$$t_{overhead} = t_{handle\ interrupt} - t_{ready\ queue} \text{ (interrupts)}$$

$$t_{access} = (1 - p) \times t_{ma} + p \times t_{page\ fault}$$

Revision Sheet (Y3S1) – Concurrent Systems

1. Syntax & Concurrency Basics

Command	**Operation**
#include <iostream>	Header file to allow the use of << cout/cin
#include <thread>	Header file to allow the use multithreading
#include <mutex>	Header file to allow threads run one at a time Any thread which has to execute some lines of code which should not be modified by other threads at the same time, has to first acquire a lock on a mutex (clutching the door handle of the booth). Only then will a thread be able to run those lines of code (making the phone call). Once the thread has executed that code, it should release the lock on the mutex so that another thread can acquire a lock on the mutex.
#include <windows.h>	Header file to allow access through Windows API
using namespace std;	Standard C++ package to always be included
::	Scope resolution operator (helps to identify and specify the context to which an identifier refers, particularly by specifying a namespace)
std::cout << "Hello World" << endl	Prints Hello World to the console followed by a newline
cout << i;	Prints contents of i to the console (std:: not always required)

void f1()
{
   std:cout << "Hello";
}
	Basic thread structure

int main()
{
   std::thread t(f1);
   t.detach();
}
	Basic main structure attaching/deteaching thread 't'. Must be int as you pass a thread to it.

std::thread t(f1);	Declares thread instance 't' which runs function 'f1'

std::thread t[5];
t[0] = std::thread(run, 0);
	Declares an array of 5 threads and begins the first one

t.detach()	Main continues to run while allowing thread 't' to run in background (until t finished)
t1.join()	Main waits for thread 't1' to finish before proceeding

class ShObject
{
      public:
            ShObject(int value)
            : x(value) {};
//equiv to x = value;
        int getX()
        {
            return x;
        }
	Class declaration to build objects
	
	Access specifier - variables visible to all that are aware of ShObject

`void setX(int value) { x = value; } private: int x; };`	Access specifier - variable local to ShObject
`: x(value) {};`	Parsing values into public variables (x = value)
`ShObject aSO(1);`	Creates instance ‘aSO’ of ShObject
`aSO.setX(i);`	Calling setX function of aSO instance of ShObject
`std::this_thread:sleepfor (std::chrono:milliseconds(100));`	Delays by 100ms
`class SharedObj { public: SharedObj(int value) : x(value) {};`	Constructor - when called, creates an instance of the object. Must have the same name as the object, used to pass input values
`ShObject ShObjectinstance(0)`	Uses constructor to make new instance of object and pass it its arguments
`&obj`	In C++ (but not C) the ampersand is a reference parameter. It passes a reference to the argument, similar to passing a pointer, except that you don't need the pointer dereferencing syntax when you use it. **Without** the reference parameter, many things are **passed by value**, which means for some complex data types a copy is made, and your changes don't affect what was actually passed in. **With** the reference parameter (&) however, the original **object passed in will be modified** by anything done to it in the function.
`std::thread t1(&Account::deposit, &acct, 500)`	Pass by value avoided, &Account object will be modified as well as the &acct value so will look different globally.
`try { getdata(); } catch (…) { cout << “Error” }`	Exception handling in C++
`std::thread(&TClass::run, std::ref(tc));`	A class template that wraps a reference in a copyable, assignable object. It is frequently used as a mechanism to store references inside standard containers (like std::thread) which cannot normally hold references.
`TClass(Buffer &b): buffer1(b) {} private: Buffer &buffer1;` *In main:* `Buffer bufferMain; TClass tc(bufferMain);`	Constructor used to pass object from another class into current class. Passed object must be declared privately to avoid being globally modifiable. Object must be declared in main.

std::mutex m;	Sets up the mutex called ‘m’
m.lock();	Thread locks onto the mutex
m.unlock();	Thread unlocks from the mutex
std::lock_guard<std::mutex> guard (mu)	Sets up the lock guard ‘mu’. Whenever lock_guard goes out of scope the mutex will always be unlocked (prevents exceptions to be thrown)
std::unique_lock<std::mutex> locker (mu)	Sets up the unique lock ‘mu’. More flexible than lock_guard as allows mutex to be locked and unlocked an arbitrary number of times.
while (mutex_locked);	Busy-waiting (or polling) approach putting thread in a spin-lock loop
wait()	Suspension. Calling thread is removed from processor.
notify_one()	Resumes one thread at random waiting on the mutex.
notify_all()	Resumes all threads waiting on the mutex.
std::condition_variable cond_var	Sets up the condition variable cond_var.
cond_var.notify_one();	Uses the condition variable to notify one.

```
#include <mutex>
#include <condition_variable>

class Semaphore
{
public:
    Semaphore (int count_ = 0)
        : count(count_) {}

    inline void notify()
    {
std::unique_lock<std::mutex>
lock(mtx);
        count++;
        cv.notify_one();
    }

    inline void wait()
    {
std::unique_lock<std::mutex>
lock(mtx);

        while(count == 0){
            cv.wait(lock);
        }
        count--;
    }

private:
    std::mutex mtx;
    std::condition_variable cv;
    int count;
};
```

Semaphore is signalling mechanism which allows a number of thread accesses to shared resources. It is not owned.

```cpp
#include <list>
#include <thread>
template<class item>
class channel {
private:
  std::list<item> queue;
  std::mutex m;
  std::condition_variable cv;
public:
  void put(const item &i) {
    std::unique_lock<std::mutex>
lock(m);
    queue.push_back(i);
    cv.notify_one();
  }
  item get() {
    std::unique_lock<std::mutex>
lock(m);
    cv.wait(lock, [&](){ return
!queue.empty(); });
    item result = queue.front();
    queue.pop_front();
    return result;
  }
};
```

`Channel is a form of synchronous message parsing (an alternative to mutex, condition variables and semaphores). The C++ implementation is shown aside.`

Concurrent system: complex group of software components and/or hardware in operation at the same time dedicated to a single application (non-deterministic due to interleaving - producer consumer problem, dining philosopher problem & reader writer problem)

Inherently concurrent system: system must handle stimuli from external world generated simultaneously (DBs, OS, ES)

Potentially concurrent systems: divide large problem into smaller parallel tasks (searching, AI, STEM)

Process: run time instance of a program (contains object code, library code, memory, access to CPU & devices…)

Stack frame: memory block large enough to store local variables

Race condition: processes race to update the values of a shared variable causing unpredictable behaviour

Atomic action: single indivisible action

Entry/Exit protocol: condition which allows or rejects entering of a critical section

Mutual exclusion: a property of concurrency control with the purpose of preventing race conditions - requires that one thread of execution never enter its critical section while another concurrent thread of execution enters its own critical section.

Condition synchronisation: a mechanism that protects areas of memory from being modified by two different threads at the same time `using wait()`, `notify_one()` and `notify_all()`.

Semaphore: a concurrency construct implemented as an object to handle mutual exclusion (not natively found in C++).

Message passing: can be done by **synchronous send** (channel || **rendezvous**), **asynchronous send** (signal || **mailbox**).

Rendezvous: used for client -server interaction where client makes request, suspends while waiting, receives reply & reruns.
Mailbox: if mailbox full suspend sender || return immediately || overwrite. Sender sends and neither Tx or Rx wait.

CSP/Occam model: processes communicate via intermediary channels.

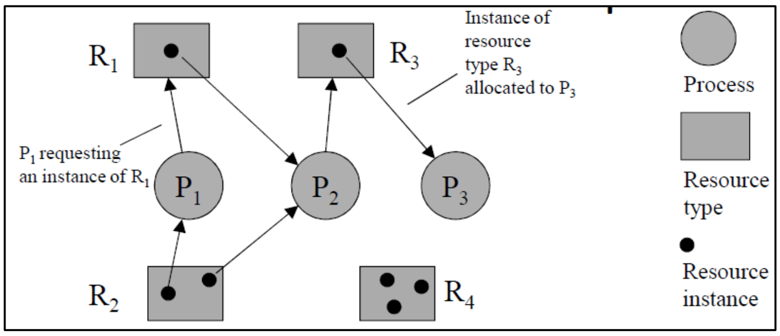
2. Deadlock

Synchronization primitives (e.g. **mutex**, **conditional variables** and **semaphores**): **simple software mechanisms** provided by a **platform** (e.g. OS) to its users for supporting **thread or process synchronization**. Usually built using **lower level mechanisms** (e.g. atomic operations, memory barriers, spinlocks, context switches etc). Synchronisation primitives can lead to **suspension** or the problem of **deadlock**.

Deadlock: set of processes become permanently blocked unable to progress, usually because shared resources are held up by another process

Resources (2): anything needed by a process to proceed, two types - **serially reusables (**e.g. CPU, memory, devices, files, buffers) & **consumables** required by process consumed and never released e.g. input data, messages, signals. **Serially reusables** only are considered as they are the type that causes deadlock. These are requested, used and released.

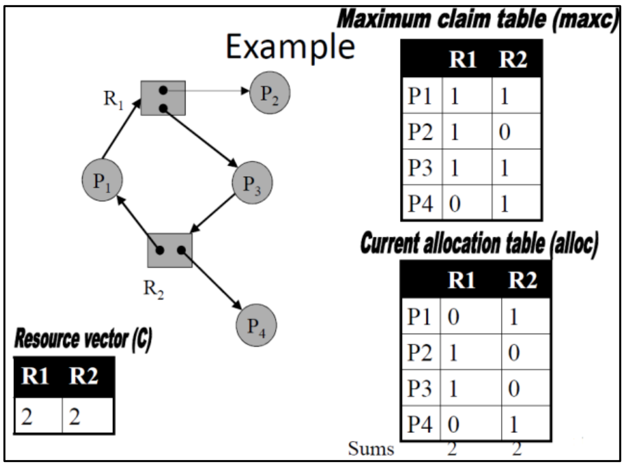
Conditions for deadlock (3) (necessary & sufficient, true IFF): processes share resources in **mutual exclusion**, process **hold resources already granted**, **system cannot pre-empt resources** from processes, **circular dependency**.


R$_1$
R$_3$
Instance of resource type R$_3$ allocated to P$_3$
Process
P$_1$ requesting an instance of R$_1$
P$_1$
P$_2$
P$_3$
Resource type
Resource instance
R$_2$
R$_4$

Resource Allocation Graph (RAG): shows processes and resources, including resource instances and their allocation. Cycles indicate *potential* deadlock. In a RAG with a **single instance** of each resource, a cycle indicates **certain deadlock**. In a RAG with multiple instances, cycle is necessary but not sufficient for deadlock.

Strategies of deadlock avoidance (3): **deadlock prevention** (system structured to make deadlock impossible - can avoid using mutual exclusion by faking e.g. spooling queues || pre-empt resources by aborting or number resources with an id & allocate in fixed order), **deadlock detection/recovery** (required some overhead - spot that a deadlock has occurred using a RAG and deal with it by aborting deadlocked processes || deallocating processes one by one until deadlock is broken), **deadlock avoidance** (anticipate deadlock when granting resource requests and take evasive action by not granting request if required e.g. **Banker's algorithm**).

Banker's algorithm: when created **each resource must state maximum amount** of each resource to be used during execution from OS **resource manager**, **allocation will occur if total resource** already allocated and requested is **less than maximum claim && system remains in safe state** (at least one sequence of resource allocations/deallocations can occur) after allocation.



Maximum claim table (maxc)

	R1	R2
P1	1	1
P2	1	0
P3	1	1
P4	0	1

Current allocation table (alloc)

	R1	R2
P1	0	1
P2	1	0
P3	1	0
P4	0	1
Sums	2	2

Resource vector (C)

R1	R2
2	2

(see examples & implementation)

Livelock: set of processes executing but not making any progress

Starvation: subset of processes unable to make progress because they cannot access needed resources, as other processes are favoured by resource allocator (maybe due to their higher priority).

Liveness & safety (2): concurrency issues are **breaches** of liveness & safety. **Safety properties** must *always* hold for nothing bad ever to happen (e.g. for a system to be safe mutual exclusion is upheld, deadlock is absent). **Liveness properties** must *eventually* hold (absence of livelock, deadlock, permanent blocking or starvation) for system to make progress.

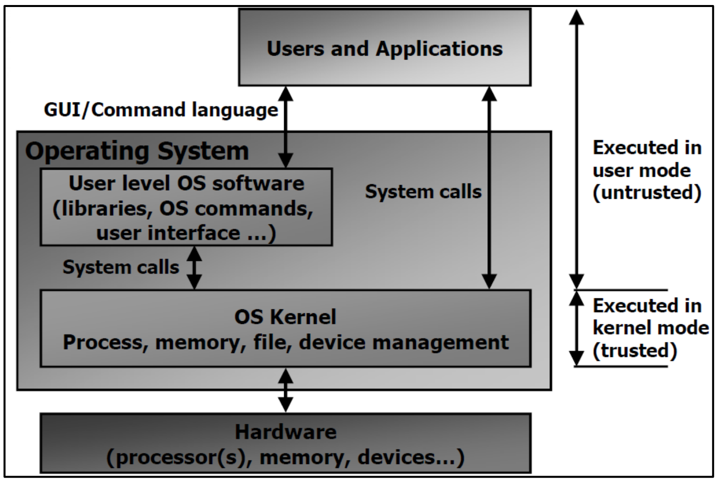
3. Operating Systems

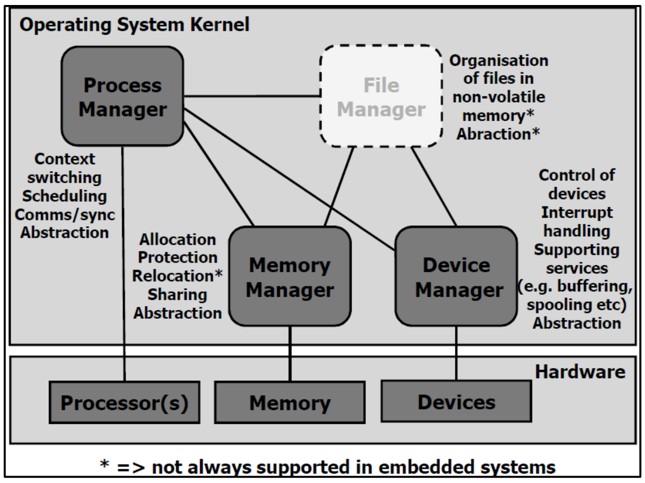
Platform = Hardware + OS, here assume 1 CPU, single core. Shell = program for terminal cmd.

OS views (2): **beautification** (makes system higher level & easier to use/program increasing productivity && providing a VM), **resource allocation** (manages resources provided by hardware fairly, efficiently and securely).

OS services (4): **process management**, **device management**, **memory management**, **file management**.

OS functions (4): **GUIs/shells**, **libraries**, **security**, **networking**


Users and Applications
GUI/Command language
Operating System
User level OS software (libraries, OS commands, user interface ...)
System calls
System calls
OS Kernel
Process, memory, file, device management
Hardware (processor(s), memory, devices...)
Executed in user mode (untrusted)
Executed in kernel mode (trusted)


Operating System Kernel
Process Manager
File Manager
Organisation of files in non-volatile memory* Abraction*
Context switching Scheduling Comms/sync Abstraction
Allocation Protection Relocation* Sharing Abstraction
Memory Manager
Device Manager
Control of devices Interrupt handling Supporting services (e.g. buffering, spooling etc) Abstraction
Hardware
Processor(s)
Memory
Devices
* => not always supported in embedded systems

<u>Trust (2)</u>: **user mode** (switches in and out of kernel mode when needed) & **supervisor/kernel mode** (admin in Win || root in Unix)

<u>Processes</u>: dynamic concurrent activities that execute a program (static). A process is an **instance of a program in execution**. Processes **hold resources** such as CPU, memory, files, devices, code… Several processes can be executing the same program at the same time (like an editor or compiler in multi-user system, like musicians playing from a score). **Can contain threads** which may or may not be known to the OS.

<u>OS services to run a program</u>: create a process → **information about process recorded**, **memory allocated**, program load module must be **loaded into memory**, process must be **scheduled** for execution.

<u>OS services during execution (2)</u>: **resource access requests** (files, I/O devices), make **system calls via API** (read/write, create new process, create a window, wait on semaphore, open/close files…)

<u>Application Programming Interface</u> (API): High-level interface into OS, allows programs to request OS services & underpins user view of system (through GUI || shell, invoke OS services via **system calls**)

Process Management		
Posix	**Win32**	**Description**
Fork	CreateProcess	Clone current process
exec(ve)		Replace current process
waid(pid)	WaitForSingleObject	Wait for a child to terminate.
exit	ExitProcess	Terminate current process & return status
File Management		
Posix	**Win32**	**Description**
open	CreateFile	Open a file & return descriptor
close	CloseHandle	Close an open file
read	ReadFile	Read from file to buffer
write	WriteFile	Write from buffer to file
lseek	SetFilePointer	Move file poi nter
stat	GetFileAttributesEx	Get status info
Directory and File System Management		
Posix	**Win32**	**Description**
mkdir	CreateDirectory	Create new directory
rmdir	RemoveDirectory	Remove *empty* directory
link	(none)	Create a directory entry
unlink	DeleteFile	Remove a directory entry
mount	(none)	Mount a file system
umount	(none)	Unmount a file system
Miscellaneous		
Posix	**Win32**	**Description**
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Change permissions on a file
kill	(none)	Send a signal to a process
time	GetLocalTime	Elapsed time since 1 jan 1970 18

<u>Protection</u>: achieved by preventing applications from **directly controlling hardware** via **privileged instructions** from processor instruction set **reserved for use by OS** (e.g. enable/disable interrupts, context switch, memory protection register access, device control, processor operation control. Also through **supervisor/kernel** and **user** mode (kernel only for **trusted** software).

<u>Interrupts</u>: used to enter kernel mode (interrutps needed for direct interaction with hardware), in kernel mode **PC & status register** (at least) **saved by hardware**, OS entry forced by ISR diverting flow of control into OS code. **Interrupt mechanism generalised** for **standard entry into OS**, achieved via **exceptions**.

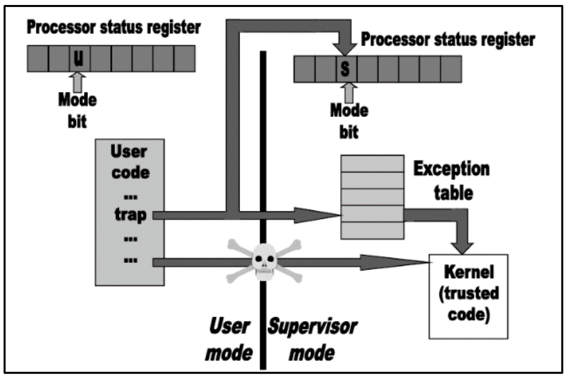
<u>Exceptions</u>: processor signals often caused by **errors** detected by hardware (division by 0, illegal bit patter in op code, memory protection error, privilege violation). Raising exception equivalent to interrupt - switch to kernel mode & save PC/SR & jump to ISR.

Exceptions are synchronous with program execution, interrupts are asynchronous (occur at unpredictable times).

<u>Traps</u>: when user code makes a system call, mode switch achieved through a **trap instruction**. Executing trap instructions (or any privileged instruction) in user mode causes an **exception**, thus mode is changed & control is transferred to appropriate ISR.

<u>Mode switch</u>: occurs due to a **system call** (trap instruction → exception) or **interrupt occurs**.

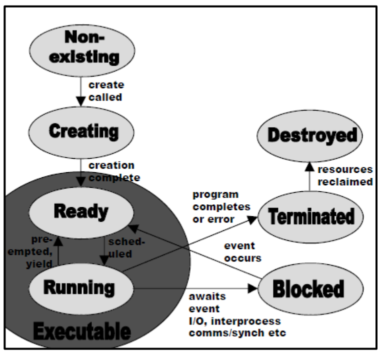
Memory is split in user space & kernel space (in kernel mode all areas of memory can be accessed, separate run-time stack for OS use in kernel space). User mode can only access user space.



3.1. Process Manager (PM)

Involves **process creation/destruction**, **context switching**, **suspension/resumption** & **inter-process comms/sync**.

Provides (4): **process/thread abstraction**, **CPU sharing abstraction**, coordination of **resource allocation**, **comms/sync mechanisms** (e.g. semaphores)

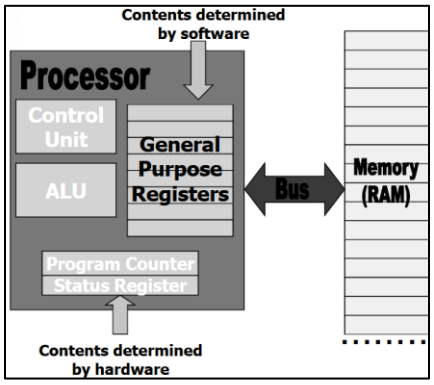


Tasks (4): **creation/termination of processes**, **sharing processor**, deciding which process to run next (**scheduling**), **inter-proc comms/sync**

API calls (2): `CreateProcess()` [Win], `fork` [Unix]. The **parent** process calls `fork` which creates a **child** process that executes the same code as parent but in a different address space. To get child to execute a different command, must use `execve()` which specifies the pathname of executable code of new program.

Only one process can be executing on a CPU at any time, PM shares CPU between processes (processes temporarily cannot execute when **blocked** to enable other processes to make progress). This required context switching.

Process switching (or context switch): act of **stopping** and executing process and **resuming** a different one. Context is the information about a process needed for full restoration (usually stored in **Process Control Block (PCB)** data structure). Runs in **supervisor mode** enabling **privileged instructions** to be executed (required to change PC & disable interrupts to prevent interference/race conditions within PM).


Contents determined by software
Processor
Control Unit
ALU
General Purpose Registers
Program Counter
Status Register
Bus
Memory (RAM)
Contents determined by hardware

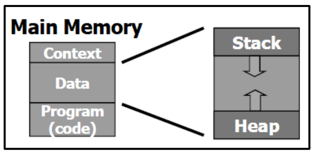
Process Control Block (7): **process ID**, **process state** (new, ready, running…), **process priority** (if in use), **registers** (address of next instruction to be executed & other register contents), **memory info** (stack pointer, MMU registers, page table…), **I/O info** (devices alloc to process, list of open files), **pointer**.

Scheduling: act of **deciding** which process to run next

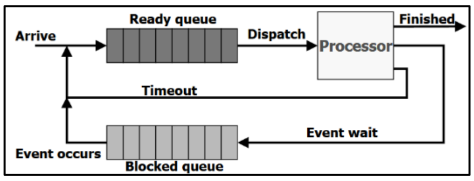
Dispatching: act of **restoring** context of a scheduled process so that it can execute

Registers: used to store data (intermediate results) from processor instructions (in load/store architectures) which must be saved to memory during context switches as they contain state information and will be overwritten if process switched.

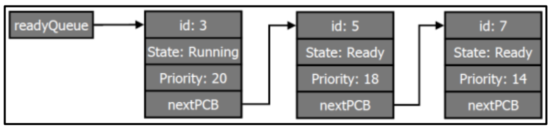
Run-time stack & heap: area of memory allocated by a process executing, **not required to be saved during context switch** as memory manage must **ensure no other process uses this area of memory**. Stack sets memory aside for static allocation (LIFO), head sets memory aside for dynamic memory allocation (no order of retrieval).



Process Manager: uses data structures to hold PCBs (implemented as a struct/object) of ready, suspended or running processes && state lists/queues (implemented as linked-lists) to manage system processes.

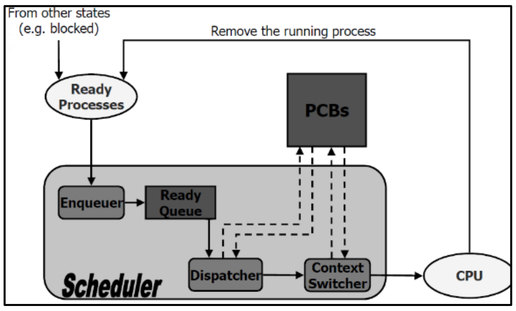


Process Manager Queues: only one process executing at any instant (single proc, single core architecture) so other processes must be ready in a queue. In multi-processor systems, number of processes > number of processors.



Multiple blocked queues can occur due to difference causes of blocking. Ready queues are implemented by pointers.

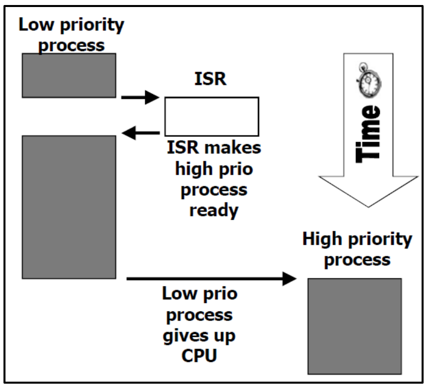
Process switches can occur when process moves from **running** to **blocked**, **running** to **ready**, **blocked** to **ready** or because of **system calls** or error conditions.



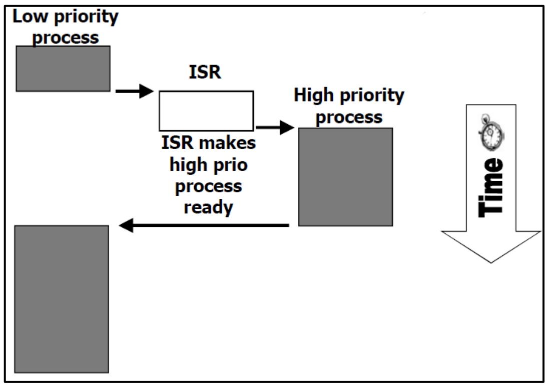
The scheduler: operation is shown aside - **enqueues**, **dispatches** & **context switches**

Voluntary Processor Sharing: **non-pre-emptive scheduling** (once a process starts to execute it continuous until **termination** || **voluntary relinquishing** of processor via `yield` system call where application tells OS it no longer wants to continue) & **cooperative multiprocessing** (process gives up processor to enable others to execute. If a process does not execute a yield because it may be stuck in an infinite loop, system reboot required).

Involuntary Processor Sharing: **pre-emptive scheduling** in use (kernel/hardware forces process to relinquish processor via **interrupts**, usually programmable timers, after which interrupt handler passer control to scheduler to decide which process to run next).



Low priority process
ISR
ISR makes high prio process ready
Time
High priority process
Low prio process gives up CPU



Process Selection: many possible criteria are valid & these are dependent on **objectives of OS**.

Objectives of OS (4): ensure important processes executed within a fixed time-frame (**real time OS**), minimise time to respond to user (**interactive OS**), maximise number of processes completed per second (**transaction processing with large databases**), ensure all processes are treated equally (**time-sharing OS**).

OS performance metrics (5): **maximise processor utilisation** (keep processor busy with useful work for as long as possible i.e. 100% CPU), **maximise throughput** (measure of useful work accomplished), **minimise turnaround time** (time between process submission & completion), **minimise waiting time** (sum of periods spent by process on the ready queue, scheduling does not affect execution or IO time but does influence waiting time), **minimise response time** (time from submission of a process to its first response).

Scheduling algorithms (4): **first come first served FCFS** (as processes arrive in the system they are added to the end of the **ready queue**, processes run to completion thus **non-pre-emptive** with **few context-switches** necessary - efficient as low switching overhead but throughput & responsiveness often poor), **shortest job first SJF** (**assumes execution time is known** which can be hard, runs process with shortest execution time first, optimal for **minimum average waiting time**, can be **pre-emptive** interrupting running process with a shorter one || **non-pre-emptive**), **round robin RR** (often used in interactive multi-user OS, CPU time divided into time slice with each process on ready queue getting equal time slice in order of arrival & **pre-empted** as required, performance **highly dependent on size of time slice** as **larger** time slice **approximates to FCFS** & **small** time slice has **large switching overhead**), **priority based PB** (**pre-emptive** scheduling based on priority number allocated to process, often used in real time OS).

Scheduler execution: occurs only when a process is **created** || **terminated**. This is done when **interrupt** occurs after ISR executed a **system call** is made for OS to enter scheduler.

Multi-level queues: priorities are not usually unique (several processes have same priority) so **multiple ready queues** are used **one for each level**. Within a priority level any scheduling algorithm can be used but RR is most common (or variants thereof - win 32 priority levels with RR in each & BSD Unix too).

Traditional processes: a **sequential** program in the course of execution which will involve extensive **time consuming context switching** if run concurrently due to changing memory, devices, files, OS…Overhead discourages concurrent systems where processes co-operate in a single application since every time a process is suspended a costly context switch is required. But this can be fixed with **threads**.

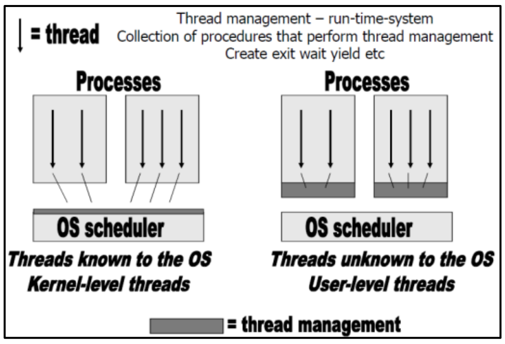
Threads: exploits concurrency without high overhead as it allows concurrent activity **within** traditional OS processes (threads have PC, stack, register set…). This is achieved by allocated resources to the process which are shared by the thread it contains (files, devices, data section…) thus thread switching is fast as execution context only requires small changes. Processes with threads are known as **modern processes**.

Per-process Items	**Per-thread Items**
AS, global variables, open files, open devices, child processes, signals & signal handlers	PC, registers, **stack**

Global variables: **extern variables** in the context of C

<u>Child process</u>: **process created** by another (**parent**) **process**

<u>Signals</u>: an **asynchronous communications mechanism** between processes (when process receives signal it is forced to execute **special signal handling function**)


= thread
Thread management – run-time-system
Collection of procedures that perform thread management
Create exit wait yield etc
Processes
Processes
OS scheduler
OS scheduler
Threads known to the OS
Kernel-level threads
Threads unknown to the OS
User-level threads
= thread management

<u>Kernel-level threads</u>: **Known to OS** (in Windows), OS performs all scheduling, **thread is unit of scheduling**, **process is unit of resource allocation**.

<u>User-level threads</u>: **Unknown to OS**, **library package provided** to manage threads in context of enclosing process (pthreads), low overhead, **OS sees processes as single threaded** where **process** is the **unit of resource allocation & scheduling**.

<u>Advantages of user-level threads (3)</u>: **speed** (much less context to save, thread management does not cause entry to OS, simple function calls faster than system calls), **customised thread scheduling algorithm** (rather than relying on OS), **scalability** (data structures used to manage threads, rather than centralised in OS PM).

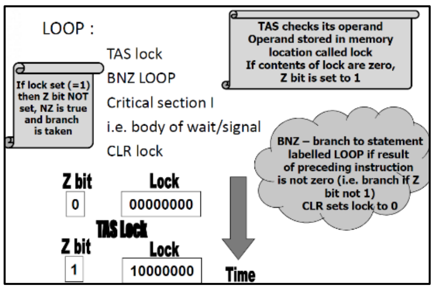
<u>Disadvantages of user-level threads (2)</u>: **blocking system calls** (if user-level thread makes a blocking system call this has **unpredictable behaviour on enclosing process** - think colour changing assignment), **no mechanism for thread pre-emption** (if OS does not know of existence of threads it cannot pre-empt those that are monopolising the processor e.g. timer interrupts cause process switches not thread switches)

<u>Synchronisation primitives</u>: implemented in **PM to allow process switches** (i.e. a **wait** operation can lead to **suspension** while a **signal** operation can lead to resumption). Allow **locking/unlocking mechanism**, **process blocking mechanism**.

<u>Semaphores</u>: useful implementation of synchronisation primitives (predates OO programming). Can be thought of as object of a class with a **non-negative integer value** (private instance variable), 2 public operations `wait()` and `signal()` and operation to initialise semaphore to a particular value (class constructor in OO). Wait and signal **must be atomic actions** (i.e. must complete without interference / cannot be interleaved with other operations on same semaphore). Both operations contain **critical sections** that must be **locked** & **unlocked** which is handled internally to OS with `lock()` and `unlock()`.

<u>Semaphore queues</u>: when a thread/process executes wait on a zero-valued semaphore it is **suspended** (context saved, removed from processor), PCB is stored **on a queue** associated with semaphore essentially part of OS **blocked queue** (necessary to allow several t/p to share a semaphore so multiple t/p can be suspended at once).

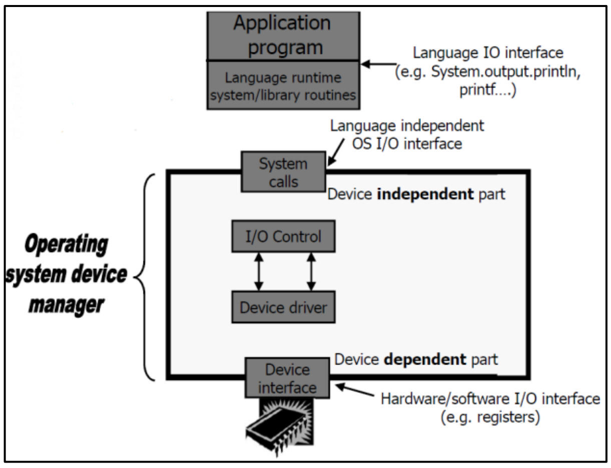
<u>Key issue & solutions (3)</u>: how to implement lock & unlock as **atomic actions** - **pure software approach** (complex Peterson, Dekker || Bakery algorithms), **disabling interrupts** (since **pre-emption** is implemented with interrupts, by disabling these we can achieve **atomicity** but only for single processor systems), **special instructions** (e.g. **Test and Set TAS** which is a machine instruction which takes 1 operand `TAS lock` and accomplishes subsequent action as one single indivisible step - but operand ends up as zero and previous operand value is stored in status register flag).


LOOP :
TAS lock
BNZ LOOP
Critical section I
i.e. body of wait/signal
CLR lock
TAS checks its operand Operand stored in memory location called lock If contents of lock are zero, Z bit is set to 1
If lock set (=1) then Z bit NOT set, NZ is true and branch is taken
BNZ – branch to statement labelled LOOP if result of preceding instruction is not zero (i.e. branch if Z bit not 1) CLR sets lock to 0
Z bit
0
Lock
00000000
TAS Lock
Z bit
1
Lock
10000000
Time

<u>Busy-waiting</u>: unavoidable status of processes which can't pass the semaphore lock. Done by executing a tight loop repeatedly examining operand (TAS lock will have a value of 0 when free). Since wait & signal are short operations this time is short thus this implementation is suitable for both single and multiprocessor machines.

	Wait and Signal	Lock and Unlock
Purpose	General process synchronisation	Mutual exclusion of processes from wait and signal operations
Implementation level	Software	Hardware
Delaying mechanism	Queueing	Busy wait / disabling interrupts
Typical delay time	O(s)	O(ms)

3.2. Device Manager (DM)


Application program
Language runtime system/library routines
Language IO interface (e.g. System.output.println, printf....)
Language independent OS I/O interface
System calls
Device independent part
Operating system device manager
I/O Control
Device driver
Device dependent part
Device interface
Hardware/software I/O interface (e.g. registers)

Like file management, OS provides simple **abstraction** for reading & writing to devices/files.

DM responsibilities: **coordinate access to shared resources** to **ensure devices used in mutual exclusion** can be used **concurrently**. Each system has a **finite number of devices** (each of these is considered a **resource**).

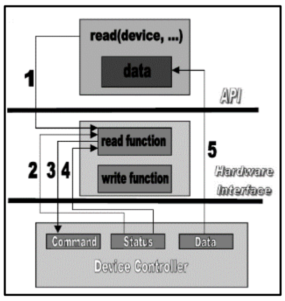
Key issue: IO devices exhibit an enormous range of data transfer rates (e.g. switches O(1), optical comms O(10^8) bps) which must be overlapped with processing (orders of magnitude faster) to ensure system performance.

Devices composed of (2): **device controller** (connects device to address and data buses, contains registers that can be accessed by software for interfacing) & **device itself**.

Device driver: software module that contains code to access the device controller registers (normally supplied with device, part of OS)

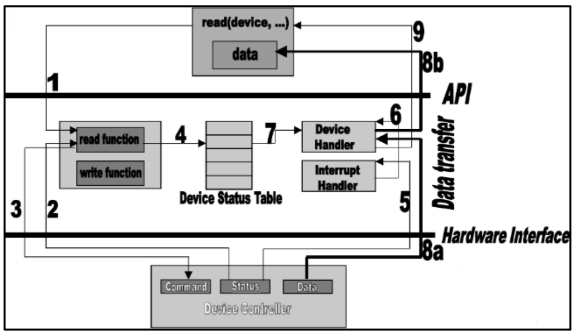
Device controller access (2): **memory-mapped** (more common e.g. in PIC - device registers appear as typical memory locations within processor address space manipulable by any memory-referencing instructions) & **IO mapped** (specialised addresses manipulable by special hardware instructions). Memory-mapped allows use of full instruction set with usual load/store instructions while IO mapped doesn't use up processor address space and simplifies address decoding.

IO techniques (3): programmed IO **polling/busy-waiting** (highly involved processor) || **interrupt driven** IO || **Direct Memory Access** (DMA, low level of processor involvement)



Programmed IO polling (5): application **requests a read**, **driver checks SR** while busy then wait, **driver writes input command** to command register, **driver repeatedly polls SR** waiting for operation to complete, **driver copies contents of data register** back to user process.

Simple, basic, efficient (3 machine instructions) but **highly inefficient** as this simple operation is repeated many times without success and is **busy-waiting**. Better for hardware to inform OS when ready i.e. interrupt.

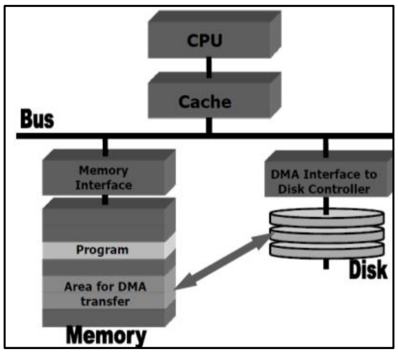


Interrupt driven IO (6): **interrupt hardware signal from device** gets attention of processor, OS **switches to kernel mode**, **PC & SR saved** on stack in memory, **PC is loaded with ISR which acquires data** from device, return-from-interrupt (**RTI**) **restores PC & SR**, **switching back to user mode**.

- total time to execute process: $t_t = t_{comp} + t_{data} + t_{overhead}$

- polling: $t_{overhead} = t_{polling}$ (repeated)

- interrupts: $t_{overhead} = t_{handle\ interrupt} - t_{ready\ queue}$

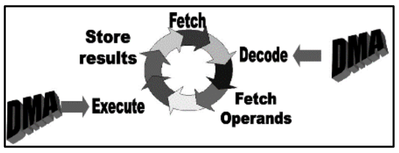


DMA: allows an entire **block of data** in/out of memory requiring a very simple **dedicated processor** (DMA device) e.g. disk controllers and network interfaces

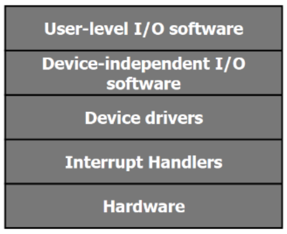
Disk controller: has control registers which allows control data to be passed by main processor such as disk start address, memory start address, amount to be transferred…

Disk controller works autonomously to transfer blocks, signals main processor when transfer complete through interrupt.

During transfer: processor executing useful processes using memory interface (DMA waiting) as soon as processor no longer uses memory interface, disk controller transfers data (when processor busy doing else e.g. instruction/operand fetch) - this is **cycle stealing**.



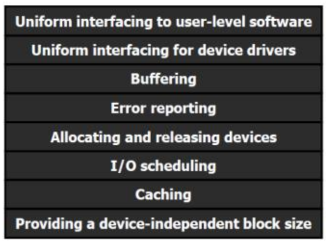
API calls: `creat()`, `remove()`, `open()`, `close()`, `read()`, `write()`, `ioctl()`



Layers of device manager (5): **user-level IO software** (not part of DM), **device-independent IO software**, **device drivers**, **interrupt handlers**, **hardware**.

User-level IO software: mostly resides within OS but some exists at **user level** in form of **library code** (IO libraries) which allow making IO system calls. Other IO software exists at user-level such as **spooling** (used to control devices that must be accessed in mutex and shared e.g. network file transfer, printers… achieved via **spooling directory** & a system process named a **daemon** - process wishing to print creates a print file in spooling directory and daemon controls the printing of the files).

Device-independent IO software: all lower layers of IO software are **device specific** while higher levels provide **IO services** (required by all devices & provide consistent interface for user-level software, common interface for device drivers).


Uniform interfacing to user-level software
Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing devices
I/O scheduling
Caching
Providing a device-independent block size

Real time OS **must support device independent IO** (**uniform set of API system calls for accessing all devices** independently from physical characteristics, must **map** these functions to the **device dependent drivers**). This relieves programmer of burden of understanding detailed operations of many different devices (faster software development).

In many Oss (both RTOS & desktop OS) device independence is facilitated using same mechanisms for file and device access (open, close, read, write…). Opening named **files** can be opening a **logical file** on a random access device or an **unstructured raw device** (e.g. serial port).

Device independent IO is possible because IO devices fall into few categories: **character-oriented devices** (1 byte at a time e.g. terminals, UARTs…), **block-oriented devices** (whole blocks of bytes are transferred e.g. HDD), **virtual devices** (streams of data processed e.g. pipes, sockets)

When application software makes IO system call, a **trap** is triggered to a kernel function enabling application to make IO request (transition from user mode to kernel mode). Thus, to add a new device to system, must edit code & recompile. **Reconfigurable device drivers** allow this to be avoided by using a `switch` statement with a **table** of drivers regularly updated on run-time via a **device registration procedure**.

IO API calls can be either **blocking** (calling application blocks, is descheduled and moved to the blocked queue (see 3.1) and resumed when IO complete) or **non-blocking** (IO call returns immediately without waiting, completion and requested data are signalled via interrupt, implemented with threads - see 3.4).

- **Buffering of IO** can improve efficiency and calling process is only awoken when buffer is full. Similarly writing character to an output device can also be buffered.

- **Error handling** useful as there are many sources of error/failure (**transient failures** i.e. overloaded network || **permanent failure** e.g. broken disk controller). Hardware indicates if IO was successful so OS can implement recovery strategy for transient failures only (often difficult & expensive). Device independent layer is responsible for a common error-reporting framework.

- **Allocating & releasing devices** done as devices must be a single process which is **controlled** via `open()` and `close()`. Alternatively, the process can be **blocked & placed on a queue** & when available remove from queue and allow it access to device (**resource sharing problem** solvable via semaphores).

- **IO scheduling** reorders IO requests in the queue to improve system efficiency / average response time (e.g. to minimise head movement and read latency 3 memory accesses are reordered by shortest seek first).

- **Caching**, as with hardware caching, a copy of needed data is read much faster from cache (e.g. HDD) as kernel maintains disk cache in memory which is checked on file request (and output accumulated in cache first enabling efficient head movement scheduling).

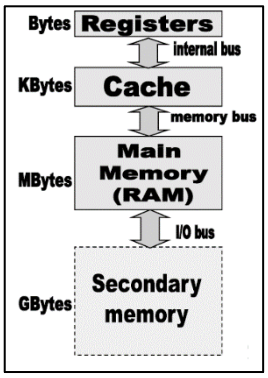
- **Device Independent Block Size** allows smaller blocks to be combined into a uniform block size for different block-oriented devices (e.g. HDD, CD, DVD…) which may differ in size.

Device drivers: device dependent software for managing device controller (part of kernel) which **can access device controller registers** by **mapping commands from device independent interface** onto specific sequences of commands for device controller. (see 3.1)

Interrupt handlers: **identity of device issuing interrupt is found** and **driver is unblocked** (see 3.1).

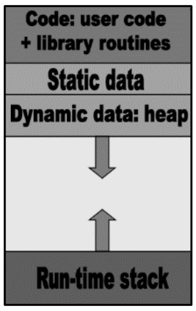
Hardware: **performs IO operation**.

3.3. Memory Manager (MM)

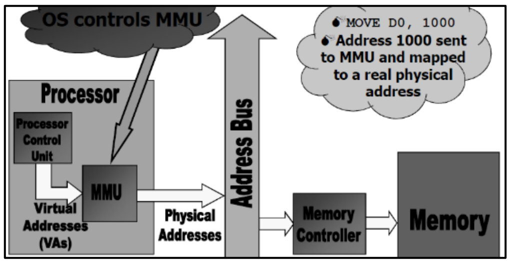


High degree of **multiprogramming** desired for **multiple processes active concurrently**. **Sufficient free memory** must be **allocated to processes** which must need memory for holding instructions, constants, extern variables, local variables… Memory organised in **virtual address space** (VAS) to allow this as well as **shared memory areas** (inter-process data transfer). **Registers** & **cache** handled by hardware, **main memory (RAM)** and **secondary memory** by OS MM.

MM responsibilities (5): **allocation** (how much memory to allocate, where to put processes in memory, how to reallocate), **relocation** (allow OS to run a process in any area of real memory), **protection** (prevent OS & hardware from accessing memory allocated to another), **sharing** (allows controlled breaking of protection among related processes), **provision of convenient memory abstraction** (enable programmers to develop code without worrying about actual amount of main memory available).

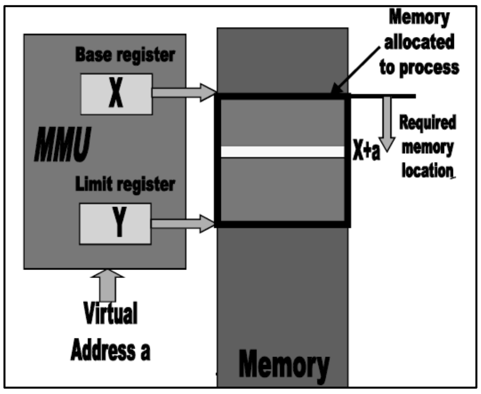


Machine instructions can only reference data in registers, cache or main memory but OS must **arrange** for instructions and data to be **transferred from secondary storage** to **main memory** transparently to user.



Virtual Address (VA): address **generated by program in CPU** (e.g. MOV D0, 1000) later **mapped by MMU** (OS controlled) to a physical address

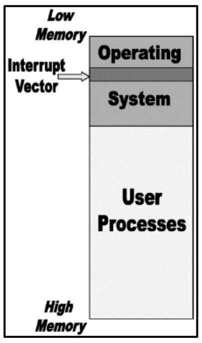
Physical Address: **actual address on the address bus**



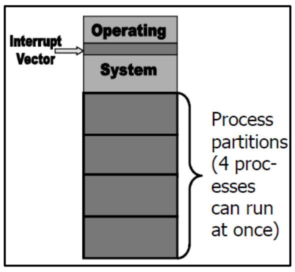
Base & Limit Registers: used by OS MM to **decide where memory processes will be loaded** (for allocation, relocation and protection).

Relocatable code: code generated by compilers to allow program variables to be **bound** to **specific memory locations** even if actual area of memory used by OS is unknown. Usually starts at memory location 0 and addresses relative to this **base address**. OS modifies these **virtual addresses** and maps to real, physical location. Programmer sees VAS, allowing relocation & protection.

VAS of a process: **range of locations** that a **processor can address** using a **fixed number of bits** (usually 32, 2^{32} memory locations ~4GiB, if more than 4GiB of RAM, 64-bit CPU is needed which allows up to 18 million TiB RAM).



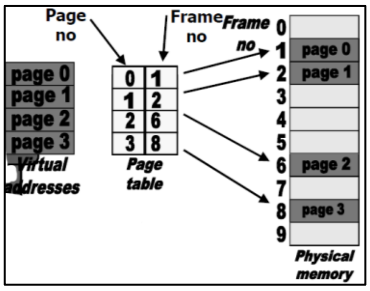
Memory allocation strategies (2): **contiguous** (simplest, memory divided into two **partitions** one for OS, rest for user processes - OS partition contains interrupt vector which resides in low memory with limit registers used to protect OS against user code), **non-contiguous multiple-partition allocation** (**static** split memory into fixed-size partitions which can contain 1 process each of limited size but that may waste excess memory - when process completes a waiting process is allocated the vacated partition || **dynamic** which improves situation but remains inflexible).



- General approach to contiguous allocation: memory **begins as a single block** (**hole**) whose size is decreased when first user process arrives. As processes arrive, they are **allocated the size they need** (MM sets base and limit for each process). As they terminate, their memory **allocation is reclaimed** by another process. **OS keeps track of allocated areas and holes** via a set of lists.

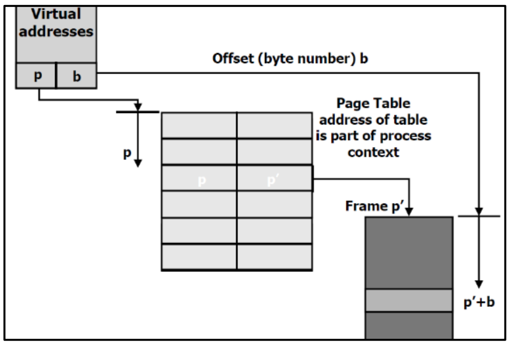
Hole allocation strategies (3): when a new process arrives, it is fitted using **list** as **best fit** (hole list sorted in order of increasing size, place process in smallest hole big enough to hold process, link remaining hole into list - leads to **smallest left-over-hole**), **worst fit** (hole list in order of decreasing size, place process in largest hole - leads to **largest left-over-hole**) or **first fit** (list holes in order of increasing base address, search until first hole big enough is found).

Fragmentation: 3 previous allocations strategies suffer from **fragmentation** as memory ends up with lots of small holes too small to hold a process individually but large enough **to hold several once merged**. Some OSes use **compaction algorithms** to move contents of memory to coalesce free holes into a large block (all processes must be relocatable).



- General approach to non-contiguous allocation: **paging** is best known technique to divide process VAS into **fixed-size pages** (512B-64KB) and **physical memory** divided into **page frames of same size** as **pages**. When a process executes, data is loaded into subsequent pages manages by a **page table**.

Page table: identifies frame in physical memory holding each page of a process VAS. **In principle**, each process requires a page table to map VAs to physical address, but this is **problematic** (for size & performance).



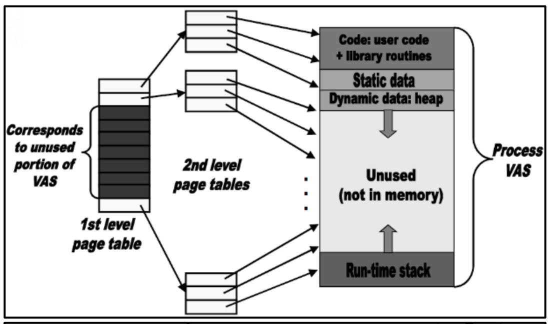
Memory reference: part of instruction that **causes a lookup** in **page table** which maps **page number** to **page frame address** or indicates page is in secondary storage. **Offset in VA added** to **page frame** address to give **physical memory address** of desired item.

VA's have 2 parts: **page number** (MSBs), **page offset** (LSBs) giving position of required address **within** page. If page size is 2^n, bottom n bits of VA represent page offset, rest represents page number. If VA is x bits VAS is 2^x *bytes*.

(**see examples & implementation**)

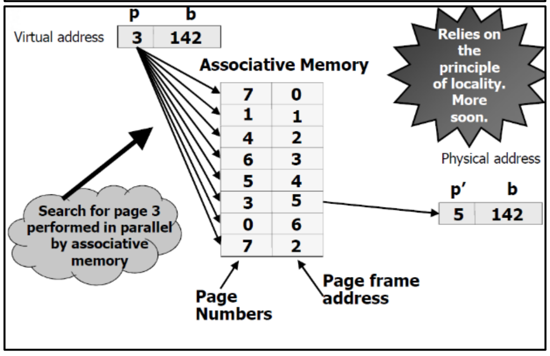
A process arrives needing *n* pages, *n* free frames required (except with virtual memory scheme), free frames allocated & pages loaded into frames, appropriate entries made in **page table** & OS updates its **frame table** (used to keep track of physical memory, one entry per frame to indicate if frame is allocated and to what process). Page table can be kept in main memory and a pointer to it is stored with process's context (loaded into a page-table base register when process runs).

Paging problems (2): **page table size** (one page table entry per page in processor AS thus 32-bit processor with 4KB pages will have a page table with 1 million entries, 4MB per page table & one page table per process) & **performance**.



Solutions to size issues (3): **keep most of page table on disk**, use a **multi-level page table structure** & use **inverted page table** (out of scope of module)

Address Translation (AT): uses hardware support to avoid compromising **system performance** since AT **doubles** number of memory accesses (access page table then access desired location). Based on **associative memory** cache using a **translation lookaside buffer** (**TLB**) which holds most commonly used subset of page table (PT). TLB entries **searched concurrently not sequentially**. If page not found in TLB, details loaded into TLB from PT prior to translation (done by MM, more sophisticated device than before).



Shared pages: **paging facilitates shared pages** (particularly code pages e.g. compiler in multi-user environment, OS page). Processes sharing code have **code pages mapped to the same page frames** which hold the code. To be sharable, **code must be re-entrant** (accessed by multiple processes concurrently - sharing processes have their own copies of data).

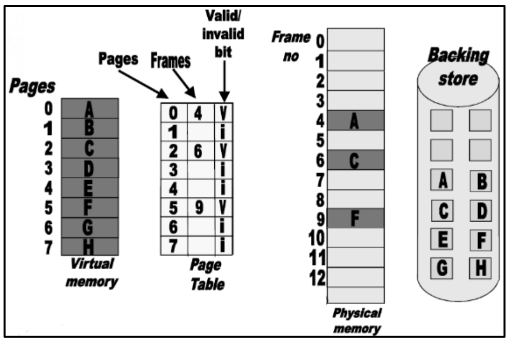
Virtual Memory (VM): an **MM approach** often **built upon a demand paging system** which **allows only a subset of a program's pages to be loaded into memory** upon **runtime** while the **rest stays on secondary storage**. This is done since **some program facilities are rarely used** (e.g. error handling routines, data structures often allocate much more memory than needed).

VM Advantages: programs to be **unrestricted by physical size of main memory** (so can code for a very large VAS), **programs to use less memory** thus **more programs can be run at once** (increase in concurrency, throughput & utilisation - no increases in response/turnaround time), **less IO needed**. Systems that execute partly loaded programs are said to have **Virtual Memory** as programmer **programs for the VAS** and the **OS with hardware assistance maps or binds these VAs** to physical addresses only a portion of which is in memory while rest is on secondary storage. (different from SWAP which is temporary memory only used when RAM fails or is full)

Demand paging: when process is run, only bring into main memory page frames the pages that will be needed soon. Pages have a valid/invalid bit which shows if page is in memory or on disk (**valid** = within process VAS & in memory, **invalid** = either invalid address || page is on secondary storage).

Has large impact on performance access time t_{access} :

$$t_{access} = (1-p) \times t_{ma} + p \times t_{page\ fault}$$



Page fault: generated if **process references a page not in memory** which enables **hardware to notice an invalid bit** in **page table entry** (during **address translation**) triggering an **exception** that **causes a trap** in OS which **initiates transfer of missing page** from disk.

Fault time includes (3): servicing interrupt - O(µs), transferring page to disk O(ms), restarting process O(µs).

Principle of locality: if a program references an area of memory it is likely to access nearby locations shortly so when lines are loaded into cache, cache misses are rare as relevant data is loaded in advance so performance is usually excellent.

Hardware support: same as for paging & hard disks - an area of HDD used for paging is SWAP space on secondary storage

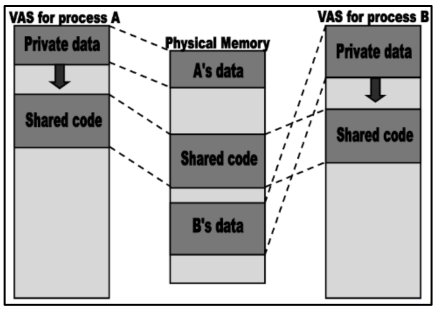
Allocation & placement policy for pages systems: easy as **page size is fixed**, for k pages simply find k free page frames (**no fragmentation** although usually **internal fragmentation** can occur as last page is on average only half full because processes are **not exact multiples of page size**). If not enough free frames, **page from another process swapped out** to make room and relevant pages are updated - this is **page replacement** (some pages locked by OS by locking bit in page table entry).

Page replacement policies (3): **first-in-first-out FIFO** (evict page that has been in memory for longest), **least recently used LRU** (evict page that has been out of use the longest), **not recently used NRU** (modified FIFO where reference bit is set every time page is written to or read, periodically cleared by OS - when page fault occurs OS checks reference bits & removes)

Page fetching policies (2): **demand fetching** (data moved from secondary storage to main memory when page fault occurs) || **anticipatory fetching** (or prefetching, requires prediction of future needs in running process & can be done by knowing program behaviour to exploit **principle of locality**).

As embedded systems become more complex, memory protection thus VA become increasingly important too. Temporal non-determinism due to page faults means process timings cannot be guaranteed so unsuitable for real-time OSes (can be solved by locking real time tasks into memory so page faults cannot occur).

Memory management alternative (1): **segmentation** (alternative approach to reflect programmer's different uses of memory for code, run-time stack, local variables of functions, heaps…). Segmentation divides segments at compile time which can implement protection, sharing, relocation, virtual memory…



Segment tables & VM: each process has a **segment table** which records location and size of segments. Different segments of a process may be dispersed (**non-contiguous**) but items belonging to same segment occupy contiguous areas of memory. Each VA is interpreted as a segment number and an offset within segment.

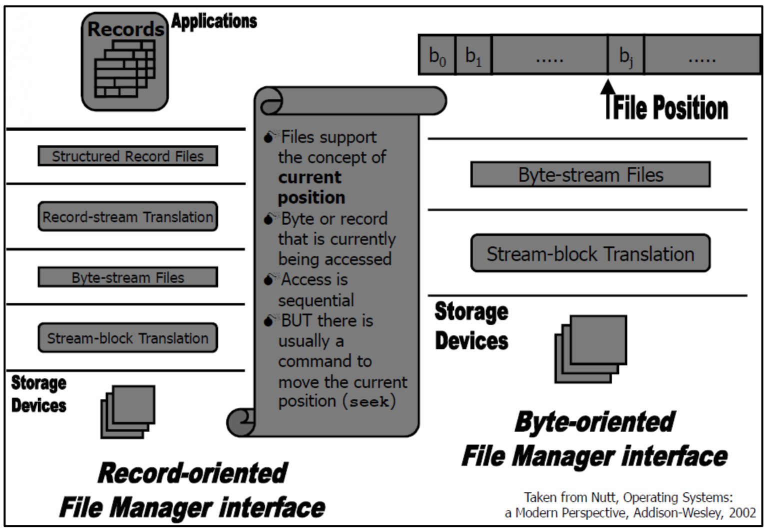
3.4. File Manager (FM)

Handled by the File Manager (FM) to manage data residing on **secondary storage** / **backing store**. (*Needs PM, MM, DM*)

FM responsibilities: provide a high-level file explorer interface, protect shared files from other users, file recovery/restoration after system crashes, mapping access requests, managing backing store (allocating/deallocating space, keep track), move file components between main memory and secondary memory.

FM interfaces with: GUI/shell (if user issues file commands), Memory Manager (transfer of files from secondary to main), Device Manager (interface with secondary storage), Process Manager (concurrent access to files typically controlled by synchronisation of primitives such as semaphores).

Files: name of organised data **permanently** stored in secondary storage (programs, documents, data, photos…). Storage device details hidden/abstracted from user/applications. Each file appears to FM as a sequence of bytes or record structures. Organised in **blocks of bytes** only addressable at the block level. Files are a set of blocks allocated to by FM. FM maps between several views of files: **collection of information** (user view), **sequence of records or bytes** (programmer view) & **set of disk blocks** (hardware/OS perspective)



- Files support the concept of **current position**
- Byte or record that is currently being accessed
- Access is sequential
- BUT there is usually a command to move the current position (`seek`)

Taken from Nutt, Operating Systems: a Modern Perspective, Addison-Wesley, 2002

FM types: **proprietary record oriented** (provides facilities for defining files as structured collections of **records** i.e. group of related data items & provides functions for manipulating records in FM) || **Win/Unix byte oriented** (byte-stream interface with functions for reading bytes from/writing to files, requires mapping byte stream to variables used in applications)

FM info maintenance: storages and manages file metadata (stored in a **file descriptor**, kept with file on secondary storage copied into OS when file is in use)

File Descriptor (FD) contents: **symbolic name** (file.txt) given by creator, **current state** (closed, open for reading, open for writing), **user** (process ID which has file open), **sharable** (can this file be used concurrently), **locks** (read/write locks), **file creator** (Thomas Hollis), **protection** (read/write/execute user access permissions), **length** (number of bytes), **creation time** (+ last modification & access), **storage device details** (device ID and address of blocks on device)

API calls: `open(filename)` checks access rights, modifies relevant metadata, returns a fileID used to refer to file while open, `close(fileID)` deletes information about file held within OS/PCB, `read(fileID, buffer, length)` copies `length` bytes from file at **current position** into `buffer` & program calling it passes it a `char buffer`, `write(fileID, buffer, length)` writes `length` bytes into file from `buffer` to current position increment current position by `length`, `seek(fileID, filePosition)` moves current position to `filePosition` for next read/write command to refer to this new position, `create(fileName)` creates a new file of given name, `delete(fileID)` deletes file & frees disk blocks & removes file descriptor.

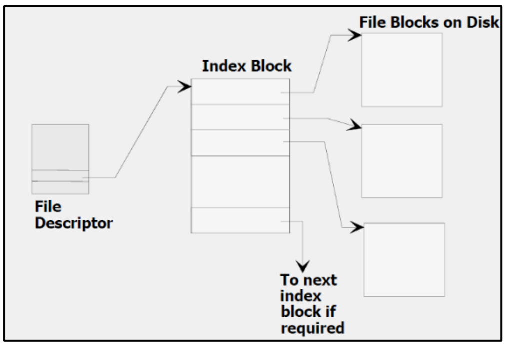
Directories: enables users and OS to keep track of contents of secondary storage in a **directory structure** (usually hierarchal), listing all files along with data about the files. Tree-like structure allows root directory to include files and references to sub-directories. File directory entry is its FD.

Storage: directories must be stored in secondary storage so that no information is loss upon machine power off (copy actively used entries into main memory for speed, keep all on secondary storage)

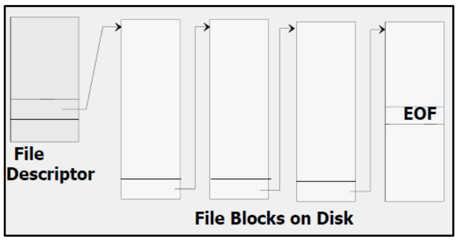
HDD: most common secondary storage divided into concentric rings (tracks) which are subdivided into sectors (each 512-1024 bytes). HDD hardware controller **access any sector by moving head**. **Disk access time** depends on **read/write time**, how far head must **move for correct track** (**seek time**), time **waiting at track for correct sector** (**rotational latency**)

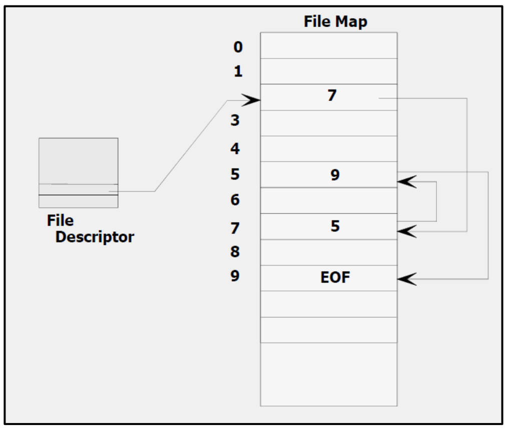
Disk Block Allocation: **FM keeps list of free blocks** in backing store, **allocates/deallocates as files grow/shrink** similarly to memory manager but with longer access times.

Contiguous allocation: Simplest allocation scheme. Files allocated to consecutive blocks, leads to **fragmentation** which needs to be regularly **compacted** making adding in the middle of a file and extending files difficult. EOF fixed.



Non-contiguous allocation: either **chaining** (linked-list approach with last few bytes of each block acting as a pointer to the next block in the file for easy moving but longer transaction number) or **indexing** (improves random access time, keeps an index block for each file in the directory entry that points to the file, look up disk block number to access a particular time, supports multiple indexing levels, effectively removes times difference between sequential and random file access)





File allocation tables (file maps): state of disk recorder in a FAT (old fashioned). Contains an entry per disk block, file directory entry contains a pointer to the FAT entry for the first block of the file which points to the second entry etc… FAT stored in secondary storage as a file, loaded into memory at book time - if disk blocks are allocated contiguously all next block pointers for a file will be contained in one FAT block so access time will be faster than simple chaining.

Free disk blocks: could be held on a list (too cumbersome), better to use a bitmap (one bit per disk block 0 = free, 1 = used/bad) typically stored on disk as a file (loaded into main memory as required).

Disk caching: reduces access time, used for reading and for writing (delayed write done at a convenient time later → may lead to information loss on system crash ∴ flushing disk cache is done regularly)