Revision notes - CS2106

Ma Hongqiang

November 26, 2018

Contents

1 Introduction to OS

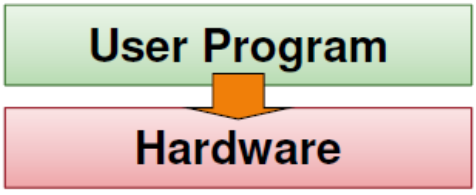
1.1 Operating System Basic Concepts

Definition 1.1 (Operating System)**.**
An **operating system**(OS) is a program that acts as an intermediary between a **computer user** and the **computer hardware**.

The evolution of OS is as below

$$\text{No OS} \rightarrow \text{Batch OS} \rightarrow \text{Time-sharing OS} \rightarrow \text{Personal OS}$$

Definition 1.2 (No OS)**.**
There is no OS for the first computer where programmes *directly* interact with hardware, and reprogramming is done through changing the **physical configuration** of the hardware.
Advantage:

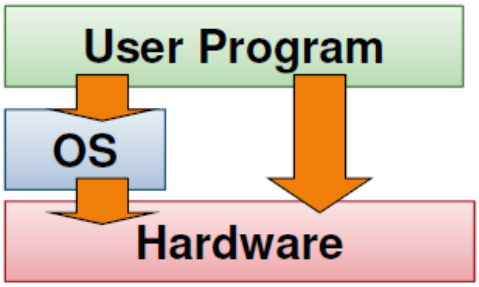


- Minimal overhead

Disadvantage:

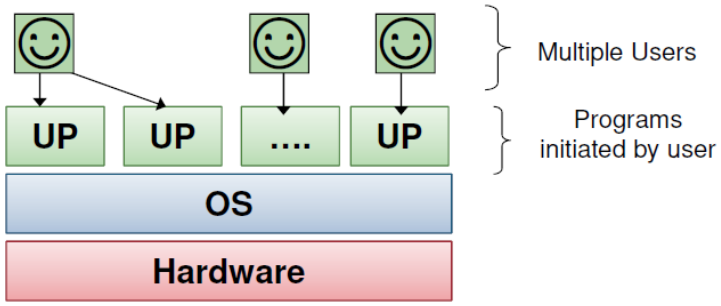
- Programmes are not portable
- Utilisation of Computing resource is low

Definition 1.3 (Batch OS)**.**
Batch OS breaks down the workflow to Input, Compute and Output, which allows pipelining to occur.
Programmes can be *submitted in batch* to be *executed one at a time*.
However, batch processing is still inefficient since the CPU will be idle when I/O. One



solution is **multiprogramming**, where multiple jobs are loaded and other jobs are to be ran when I/O needs to be done.

Definition 1.4 (Time-sharing OS)**.**
Time-sharing OS allows multiple users to interact with machine using terminals(**teletypes**). It provides user job scheduling which gives *illusion* of concurrency.
It provides CPU time, memory and storage management; essentially, it provides **virtualisation of hardware**, where each program executes as if it has all the resources to itself.



Definition 1.5 (Personal OS)**.**
Personal OS is dedicated to machine that dedicates to one user, which is not time shared. There are several models:

1. Windows Model:
   - Single user at a time but possibly more than 1 user can access
   - dedicated machine
2. Unix Model:
   - One user at the workstation but others can access remotely
   - General time sharing model

1.2 Motivation of OS

There are three main motivations:

1. Abstraction over the hardware, which has
   - *Different* capacity
   - *Different* capability, but
   - Well-defined and *common* functionality

   so that low level details can be hidden and only high level functionality is presented. It provides efficiency and portability.
2. Resource Allocator, which
   - Manages all resources such as CPU, Memory, I/O devices

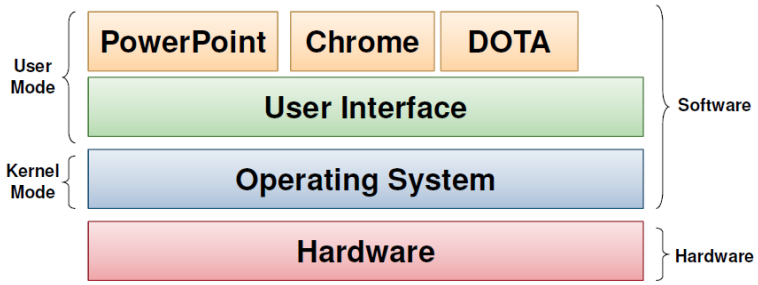
- Arbitrate potentially conflicting requests, for efficient and fair resource use

3. Control Program, which controls execution of programmes so as to
   - Prevent errors and improper use of the computer, accidentally or maliciously
   - Provide security and protection

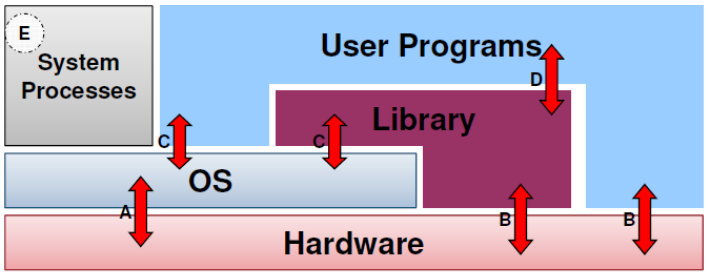
1.3 OS Structure

Operating System structure needs to impart flexibility, robustness and maintainability. Generally, the high level view of OS is that

- Operating system is essentially a **software**, which has the privilege to run in **kernel mode**, i.e., has complete access to all hardware resources
- Other software executes only in **user mode**, with limited access to hardware resources



However, one must realise the user programme can interact with hardware through OS or else. Following diagram gives you an overview:



- A: OS executing machine instructions
- B: normal machine instructions executed
- C: calling OS using **system call interface**, e.g. `fopen()`
- D: user program calls library code, e.g. `pow()`
- E: system processes, which provide *high* level services, and is usually part of the OS

In terms of functionality, OS is known as the **kernel**, which is a programme providing special features like:

- Deals with hardware issues
- Provides system call interface
- Special code for interrupt handlers, device drivers

However, kernel code has to be different than normal program as

- No use of system call in kernel code
- Cannot use normal libraries
- There is no normal I/O

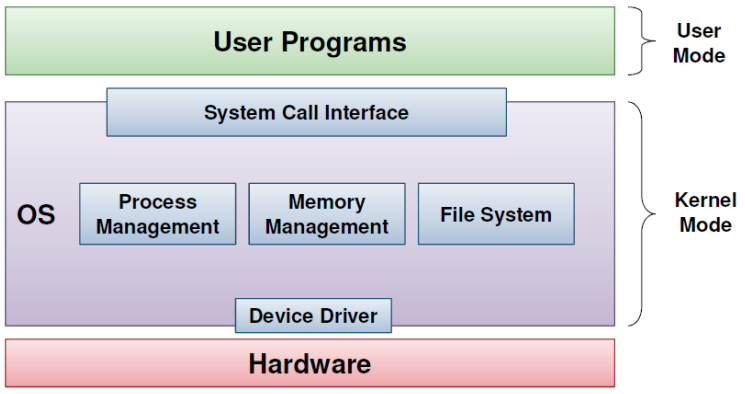
Currently, the common code organisation consists of

- Machine independent high level language(HLL)
- Machine dependent HLL
- Machine dependent assembly code

In terms of implementation, there are few ways to structure an OS, most notably monolithic or microkernel.

Definition 1.6 (Monolithic OS)**.**
Monolithic OS kernel has the defining characteristics of

- One **big** special program, where various services and compopnents are integral part.



Generic Architecture of Monolithic OS Components

Monolithic kernel has the advantage of

- Well understood

- Good performance

It has the disadvantage of

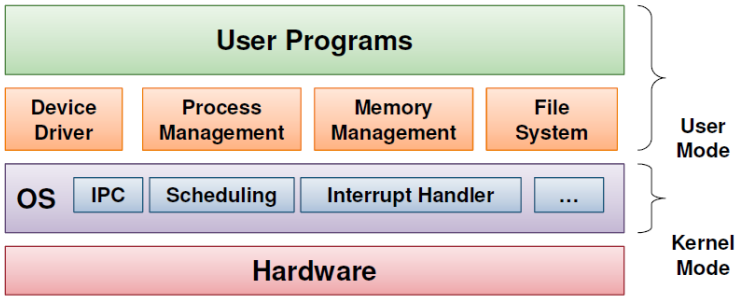
- Highly coupled components
- Usually devolved into very complicated internal structure

Definition 1.7 (Microkernel OS)**.**
Microkernel OS kernel has the defining characteristics of

- Very small and clean
- Only provides basic and essential facilities:
  - Inter-Process Communication(IPC)
  - Address space management
  - Thread management
  - etc

Higher level services like Process Management are

- Built *on top of* the basic facilities
- Run as server process **outside** of the OS
- Use IPC to communicate



Generic Architecture of Microkernel OS Components

Microkernel OS has the advantage of

- Kernel is generally more robust and more extendible
- Better isolation and protection between kernel and high level services

It has the disadvantage of

- Lower performance

Other OS structure include layered systems, client-server model etc.

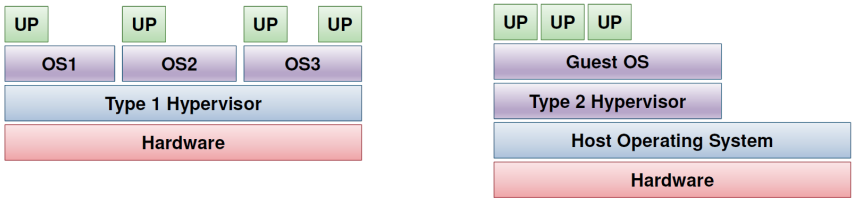


Figure 1: Type 1 Hypervisor(left) and Type 2 Hypervisor(right)

Definition 1.8 (Virtual Machine)**.**
Virtual Machine, or **hypervisor** is a software emulation(virtualisation) of hardware.

Normal(primitive) operating systems can then run on top of the virtual machine.
There are two type of hypervisor:

- Type 1: provides individual virtual machines to guest OSes
- Type 2: runs in host OS and only guest OS runs in Virtual Machine

2 Process Abstraction

As the OS, to be able to switch from running program A to B requires

1. Information regarding the execution of program A to be stored
2. Program A's information is replaced with te information required to run B

Definition 2.1 (Process)**.**
Process/task/job is a dynamic abstraction for executing program, where information required to describe a running program is contained. The information includes three components:

- Memory context
- Hardware context
- OS context

2.1 Memory Context

From `CS2100`, we know that memory contains a

- **Text** section, to store instructions
- **Data** section, to store global variables

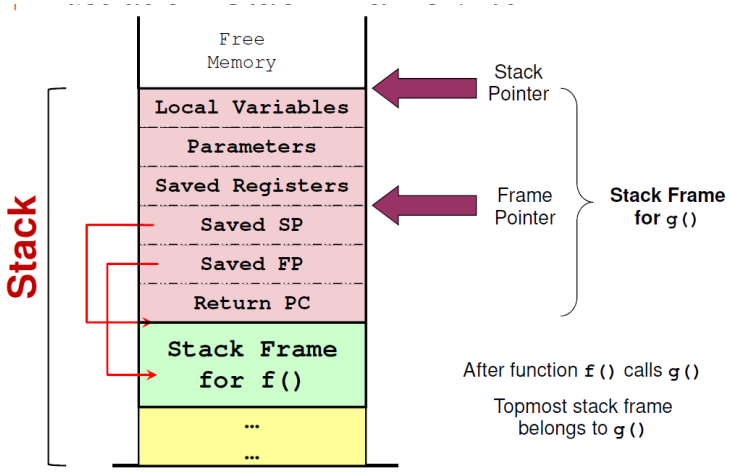
However, these two sections are unable to handle function call.

Definition 2.2 (Caller, Callee)**.**
When `f()` calls `g()`, `f()` is the **caller** whereas `g()` is the **callee**.

To handle function call, we require a new section called **Stack**.

Definition 2.3 (Stack)**.**
Stack is used to store information about function invocation. The information of a function invocation is described by a **stack frame**.
Since stack can be of variable size, the top of the stack region is logically indicated by **stack pointer**. Stack pointer usually is stored in a specialised register in the CPU. Usually, a stack farme contains the following items

- Local variables
- Function parameters
- Saved Registers
- Saved Stack Pointer
- Saved Frame Pointer
- Return Programme Counter(PC)



Definition 2.4 (Frame Pointer)**.**
Stack pointer can move during function execution. For example, the stack pointer can shift during a control statement `if`.
To facilitate the access of various stack frame items, a **frame pointer** can be used, which points to a **fixed location** in a stack frame. Other items can be accessed as a displacement from the frame pointer.

Theorem 2.1 (Stack Frame Setup/Teardown)**.**
On executing function call,

1. **Caller**: Pass arguments with registers and/or stack
2. **Caller**: Save Return PC on stack
3. **Transfer control from caller to callee**
4. **Callee**: Save registers used by callee. Save old FP, SP.
5. **Callee**: Allocate space for local variables of callee on stack
6. **Callee**: Adjust SP to point to new stack top

On returning from function call:

1. **Callee**: Restore saved registers, FP, SP
2. **Transfer control from callee to caller using saved PC**
3. **Caller**: Continues execution in caller

Remark: There is no universal way of doing function call. The above is just a example.

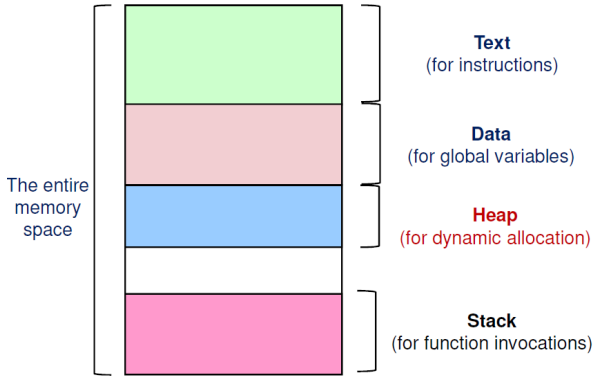
Definition 2.5 (Heap Memory Region)**.**
Heap Memory Region is a memory region that supports **dynamically allocated memory**, which is acquisition of memory space during **execution time**.

Heap memory is trickier to manage due to its nature:

- Variable Size
- Variable Allocation/Deallocation Timing

Its nature will create scenario where heap memory are allocated and deallocated in a way that creates holes in memory.

The actual memory context contains the four regions described above, namely text, data, heap and stack.



2.2 Hardware Context

In correspondence to the memory context, the hardware context contains

- General Purpose Registers
- Program Counter
- Stack Pointer
- Stack Frame Pointer

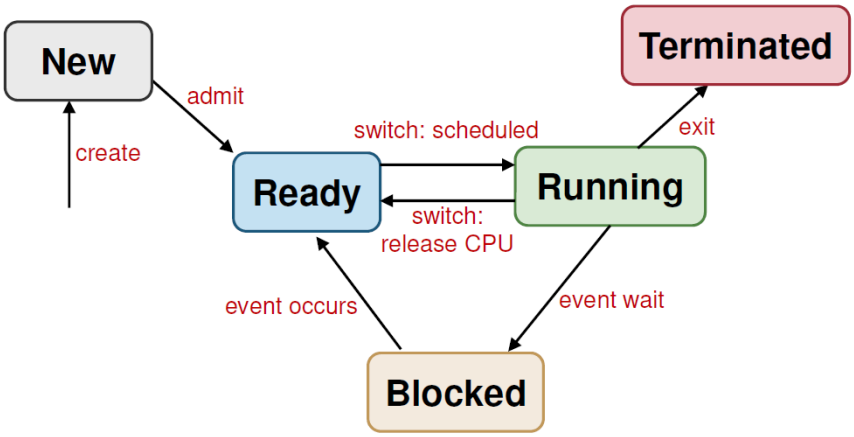
2.3 OS Context

Definition 2.6 (Process ID)**.**
Process ID is to distinguish processes from each other, and it is unique among processes.

Apart from Process ID, we also need a process state, to describe the state of the process, e.g. running or not running.

Definition 2.7 (Generic 5-state Process Model)**.**
The Generic 5-state Process Model contains 5 states:

1. New



- New process created
- May still be under initialisation, therefore *not yet* ready

2. Ready
   - Process is waiting to run
3. Running
   - Process is being executed on CPU
4. Blocked
   - Process waiting/sleeping for event
   - Cannot execute until event is available
5. Terminated
   - Process has finished execution, may require OS cleanup

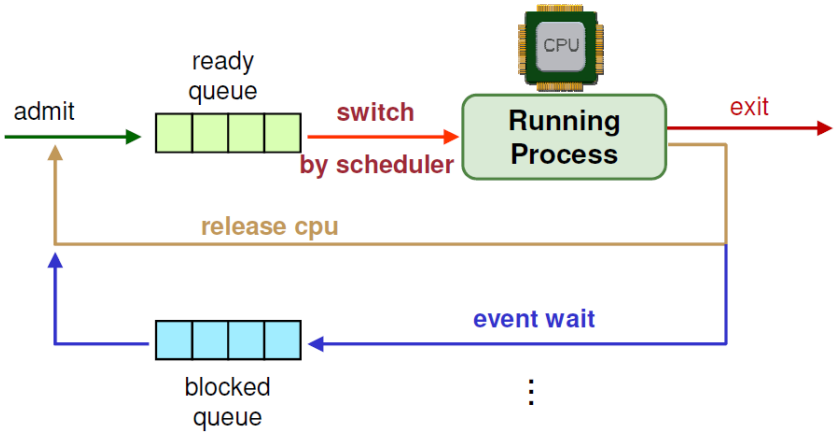
There are also different state transitions,

1. Create(NIL → New): new process is created
2. Admit(New → Ready): process ready to be scheduled for running
3. Switch(Ready → Running): process selected to run
4. Switch(Running → Ready): process gives up CPU voluntarily or **pre-empted** by scheduler
5. Event Wait(Running → Blocked): Process requests event/resource/service(system call, I/O, etc.) which is not available, or in progress
6. Event Occurs(Blocked → Ready): Event occurs, which means process can continue

In CS2106, we admit a simplified view on CPU: there is only 1 CPU, which means there is

- ≤ 1 process in running state
- conceptually 1 transition at a time

As there may be multiple processes ready for run, or awaiting resources, we use a queuing model to describe the 5-state transition. Note, here the ready queue and the blocked queue



should be viewed as **set**, not queue.

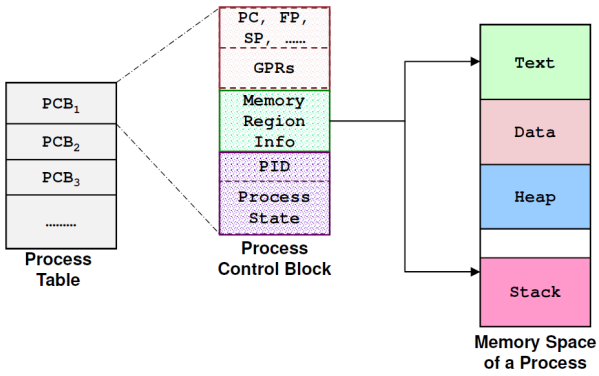
Up to this point, the OS context contains

- Process ID,
- Process State

2.4 Process Control Block

The entire execution context for a process is stored in **Process Control Block**, maintained by the kernel. The process control block(PCB) contains memory, hardware and process context.

A process Table contains process control blocks of all processes.



2.5 System Calls

System calls are Application Program Interface(API) to OS, which provides ways of calling facilities/services in kernel.
It is **not** the same as normal function call since it has to change from user mode to kernel mode.
Different OS will then have different APIs, for example:

- UNIX variant: POSIX standard, which has small number(≈ 100) of calls
- Windows family: Win API, which has ≈ 1000 number of calls

In `C`, system calls can be invoked almost directly, but the function we call is not the actual system call, as we need to have a setup before the actual call. Therefore, the function we call can be

- Function Wrapper, which is the same name and has the same parameters and the actual call, but handles the setup for caller.
  One example is `getpid()`.
- Function Adapter, which is a library function with less number of parameters and possibly more flexible parameter values.
  One example is `printf()`, which uses `write()` system call in its implementation.

Theorem 2.2 (General System Call Mechanism)**.**
Generally, system call by user programme is handled in this manner:

1. User program invokes the library call
2. Library call, usually in assembly code, places the **system call number** is a designated location, e.g., a register.
3. Library call executes a special instruction to switch from user mode to kernel mode. The special instruction is commonly known as **TRAP**.
4. Now, in kernel mode, the appropriate system call handler is determined, by using the system call number as an index.
   This step is usually handled by a **dispatcher**.
5. System call handler is executed, carrying out the actual request.
6. System call handler ended, and control is returned to the library call.
   We switch from kernel mode back to user mode.
7. Library call return to the user program, via normal function return mechanism.

2.6 Exception and Interrupt

2.6.1 Exception

Executing a **machine level instruction** can cause exception. For example, arithmetic error, or memory accessing errors are two common types of exceptions.
Exception is **synchronous**, in the sense that it occurs due to the program execution.
The effect of exception is

- Have to execute a **exception handler**. This exception handler is similar to a **forced function call**.

2.6.2 Interrupt

External events can interrupt the execution of a program. The cause of interrupt is usually hardware related, for example `Ctrl + C`.
Interrupt, unlike exception, is therefore **asynchronous**, as it occurs independent of program execution.
The effect of interrupt is

- Program execution is suspended, by executing an **interrupt handler**.

For the exception/interrupt handler, it is transferred control automatically to when an exception/interrupt occurs. Inside the handler, it performs the following routine

1. Save Register/CPU state
2. Perform the handler routine
3. Restore Register/CPU
4. Return from interrupt

After the return from handler routine, program execution will resume, and the programme *may* behave as if nothing happened.

3 Process Abstraction in UNIX

3.1 Identification

A process in UNIX is uniquely identified by Process ID(PID), which is integer-valued.

3.2 Information

A process will keep track of the information of

- **Proecss State**: Running, Sleeping, Stopped, Zombie
- **Parent PID**: PID of the parent process
- **Cumulative CPU time**: Total Amount of CPU time used so far
- etc

Unix command `ps` can extract the process information

3.3 Process Creation

`fork()` is the main way to create a new process. It returns

- `PID` of the newly created process for parent process, or
- 0 for child process

The behaviour of `fork()` is to create a new process called **child process**, which is a **duplicate** of the current executable image. The data in child is a **copy** of the parent, thus not shared. The *only* difference between the parent and child process are

- Process ID(PID)
- Parent ID(PPID)
- `fork()` return value

Note, after `fork()`, both parent and child processes continue executing after `fork()`. If we want to make parent and child processes to behave differently, we can leverage on the `fork()` return value.

3.4 Executing New Programe/Image

`fork()` is not useful if you want to execute the original process and another process which is different from the original. Suppose the another process' code is provided as a new executable, then we can use `exec()` system calls family.

Specifically, one can use `execl()` to **replace** current executing process image with a new one. This replacement will change

- Code
- Memory content

but PID and other information will still be *intact*.
`execl` takes the form of `execl const char *path, const char *arg0,...,const char *argN, NULL)`, where

- `path` is the location of the executable
- `arg0` is the executable name
- `arg1` to `argN` are vararg command line arguments, corresponding to `argv[1]` to `argv[n]`.
- The last `NULL` is used to terminate the vararg array.

Therefore, we can combine `fork()` with `exec()` to

- Spawn off a child process via `fork()`, which is replaced to the actual task process by invoking `exec()`.
- Parent process remains intact, which can continue to execute.

In fact, this way of invoking new processes is common across UNIX. Therefore, to start off, we need a special initial process, namely the `init` process, which is created in kernel at boot up time with a PID 1. The `init` process watches for other processes and respawns where needed. Other processes are created by invoking `fork()` on `init` or child of `init`.

3.5 Process Termination

To end execution of process, we can use `exit(status)`. The status will be returned to the parent process, by which this child process is created.
The UNIX convention is to `exit` with a status 0 if the termination is normal, and non-zero if the execution is problematic.
Remark: This function itself does not return.
When `exit(status)` is executed, *most* system resources used by process are released. However, some basic process resources are **not releasable**. This includes

- PID and status, which is required for parent-children synchronisation
- Process accounting information, e.g., CPU time
- Also, process table entry *may be* still needed

As most programs do not have explicit `exit()` call, return from main function will implicitly calls `exit()`, and as a result, open files will get flushed automatically, since file descriptors will be released.

3.6 Parent/Child Synchronisation

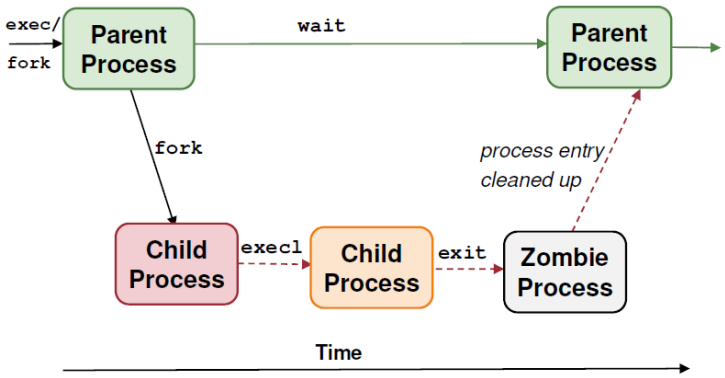
Parent process can wait for child process to terminate, by invoking `wait(*status)`. This call of `wait(*status)` will return the PID of the terminated child process. Also, the address where the `status` pointer points to will be populated with the exit status of the terminated child process. The behaviour of `wait()` is as follows

- `wait()` is blocking: parent process is blocked until at least one child terminates
- The call cleans up **remainder** of the child system resources, which includes
  - Those not removed on `exit()`, and also
  - Kills zombie process

Other variant of `wait()` includes

- `waitpid()`, which waits for a specific child proecss
- `waitid()`, which waits for any child process to **change status**

Therefore, we can summarise the process interaction in UNIX as follows: Therefore, one



point to note is that, suppose the parent does not `wait`, the zombie process cannot be cleaned up by the parent. Therefore, to handle this scenario, there are two cases to consider:

Case 1 Parent process terminates before child process:

- `init` process becomes "pseudo" parent of child processes
- Child termination sends signal to `init`, which utilises `wait` to cleanup.

Case 2 Child process terminates before parent, but parent did not call `wait`

- Child process becomes zombie, which can fill up the process table
- Therefore, we need to reboot to clear the table on older UNIX implementations.

4 Process Scheduling

We hope to achieve parallelism between processes, and therefore, **timeslicing** is used to share CPU time between processes, with OS occupying CPU between different processes to do the context switch.
In this sense, CPU is a **scheduler** deployed with certain **scheduling algorithm**. The specific scheduling algorithm will be influenced by **process behaviour and process environment**, but it should achieve a few common criteria to ensure its performance.

There are two main type of process behaviour:

- **CPU-Activity**, such as computation.
  Compute bound process spend majority of time here.
- **IO-Activity**, such as printing to screen.
  IO Bound process spend majority of time here.

There are three main type of processing environment, namely

1. **Batch Processing**, where there is *no* interaction required and there is *no* need to be responsive.
2. **Interactive**, where there are active user(s) interacting with system, therefore process should be responsive and consistent in response time
3. **Real time processing**, where there are deadline to meet.

In this course, we are only concerned about (1) and (2).

There are two criteria that is applicable for **all processing environments**:

- Fairness
  - Each process, or each user should get a fair share of CPU time
  - There should not be starvation for certain processes
- Balance
  - All parts of the computing systems should be utilised

There are two types of scheduling policies, defined by when scheduling is triggered.

- **Non-preemptive**: where a process stayed scheduled (in running state) until it blocks or give up CPU voluntarily
- **Preemptive**: where a process is given a fixed time quota to run. Process can either give up early or give up due to blocking, or be suspended by OS to give another process CPU time, if available.

The step by step scheduling algorithm is as below:

1. Scheduler is triggered.
2. If context switch is nedded, save current running process' context and place it on blocked/ready queue
3. Pick a suitable process P to run based on scheduling algorithm
4. Setup context for P
5. Run P

4.1 Scheduling for Batch Processing

Since there is no user interaction, we will try to optimise for the criteria below:

- Minimise Turnaround time, which is calculated as finish time − arrival time.
- Throughput, which is number of tasks finished per unit time
- CPU utilisation, which is percentage of time when CPU is working on a tak

4.1.1 First Come First Served(FCFS)

In **First Come First Served** scheduling,

- Tasks are stored on a First-In-First-Out (FIFO) queue based on arrival time
- Pick the first task in queue to run until:
  - Task is done
  - Task is blocked
- Blocked task is removed from the FIFO queue, and is placed at the back of queue when it is ready again

This algorithm is good as it is guaranteed to have **no starvation**, since no process can jump queue.
This algorithm is not optimal as

- Simple reordering of jobs can reduce average waiting time
- The convoy effect on processes, in which a long-running process is followed by small processes, and the first long-running process will block every short processes at the back for CPU, IO etc if they require the same resource.

4.1.2 Shortest Job First(SJF)

In **shortest job first** scheduling, after any process is finished running, we select a task with smallest total CPU time.
It is a good algorithm since it *minimises* the average waiting time.
It is not good since **starvation is possible**, as it is biased towards granting CPU to shorter jobs.
This algorithm will predict the CPU time required for a task, based on the history of the task. The common approach is

$$\texttt{Predicted}_{n+1} = \alpha\texttt{Actual}_n + (1-\alpha)\texttt{Predicted}_n$$

4.1.3 Shortest Remaining Time(SRT)

In **shortest remaining time** scheduling, when there is a job coming in, we check across all jobs, running or ready, and select the job with the shortest remaining time to be run. Since it will prompt the suspension of running process, this algorithm is preemptive.
It is good since we can give newly came shorter tasks priority to run first.

4.2 Scheduling For Interactive System

Scheduling algorithm for interactive system needs to optimise for the criteria below:

- Response time, which is the time between request and response by the system
- Predictability, which is the variation in response time

Here, **preemptive** scheduling algorithm are used to ensure good response time. For such preemptive scheduling, we need a **timer interrupt**, which triggers OS scheduler at each **interval of timer interrupt**. The system may default a value called **time quantum**, which is a multiple of interval of timer interrupt, which is the maximum execution duration of a process before it is preempted out of CPU.

4.2.1 Round Robin

In **Round Robin** scheduling,

- tasks are stored in a FIFO queue
- The first task in teh queue will be picked to run, until
  - A fixed quantum elapsed, or
  - it gives up CPU voluntarily, or
  - it blocks
- The task is then placed at the end of the queue, if ready, or placed in block queue if blocked, until it is unblocked and placed back to the end of queue

It is a good scheduling algorithm because it offers **response time guarantee**. Given n tasks and quantum q, any process will wait a maximum of $(n-1)q$ before it is executed.
However, we need timer interrupt to check quantum expiry. Therefore, there is a tradeoff:

- Big quantum: better CPU utilisation, but longer waiting time
- Smaller quantum: bigger overhead, but shorter waiting time

4.2.2 Priority Scheduling

In **priority scheduling**, each process is assigned a priority value, and we only select task with highest priority value to run upon context switch.
However, this algorithm may cause low priority process to starve. To solve this problem, we can decrease the priority of current running process after it fully occupies the time quantum.
There is another phenomenon called **priority inversion**, where some earlier ran high priority process uses resources which the next high priority process requires. This causes all of them to be blocked whereas some lower priority process which do not require the blocked resources actually gets to run first.

4.2.3 Multi-level Feedback Queue(MLFQ)

MLFQ minimises both response time for IO bound process and turnaround time for CPU bound processes.
In **multi-level feedback queue** scheduling,

- each process is assigned a priority value
- When deciding which process to run upon context switch, we pick the process with highest priority.
- If two processes' priorities are equal, run them in round robin.

The priority value is updated as below:

- New jobs will be assigned highest priority
- If a job utilise its full time quantum, its priority will be reduced
- Else, if it gives up or blocks before it finishes the time slice, it will retain current priority

4.2.4 Lottery Scheduling

In **lottery scheduling**, OS gives out "lottery tickets" to processes for various system resources, and upon context switch, a lottery ticket is chosen randomly from all eligible tickets and winner is granted the resources.
In the long run, a process holding $x\%$ of the tickets will use the resources $x\%$ of the time.

5 Process Alternative: Threads

Threads are used as an alternative to process. Process has the disadvantage of being

- **Expensive**, as we need duplicate memory space and duplicate most of the process context. It is also expensive to do context switching between different processes.
- **Hard in Inter-process Communication**, since they occupy independent memory space, therefore, there is no easy way to pass information.

Therefore, we introduce **threads** so that a multithreaded programme can have multiple threads of control, and the threads are concurrent.

Threads in the *same* process shares:

- **Memory Context**: Text, Data, Heap
- **OS Context**: Process id, other resources like files.

Unique information needed for each thread includes

- Identification, usually **thread ID**
- Registers, general purpose and special
- "Stack"

To give a comparison of context switches between processes and threads,

- Process context switch involves:
  - OS Context
  - Hardware Context
  - Memory Context
- Thread switch within same process involves
  - Hardware context only: Registers, and "Stack", which is just the frame pointer and stack pointer

Therefore, thread is much lighter than process. Essentially, threads has the following benefit:

- **Economy**: Multiple threads in teh same process requires much less resource to manage compared to multiple processes
- **Resource Sharing**: Since threads share most of the resources of a process, there is no need for additional mechanism for passing information around
- **Respoonsiveness**: Multithreaded programs can appear much more responsive.
- **Scalability**: Multithreaded program can take advantage of multiple CPUs.

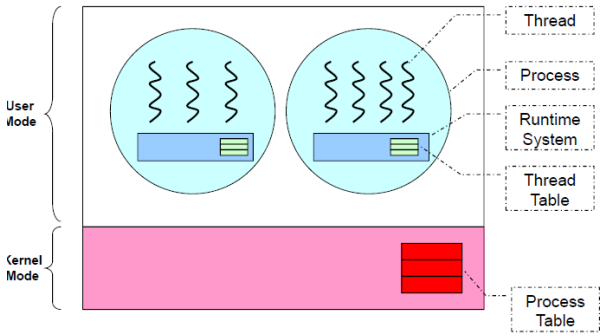
However, thread has the problem of

- disobeying system call concurrency. Parallel execution of multiple threads could result in parallel system call and correctness of behaviour may not uphold.
- confusion on process behaviour. For UNIX,
  - Call `fork()` will only fork the process of the single thread in which `fork()` is called
  - Call `exit()` will cause the whole process to terminate
  - Call `exec()`?

5.1 Thread Models

There are two ways of implementing threads, namely **user thread** and **kernel thread**.

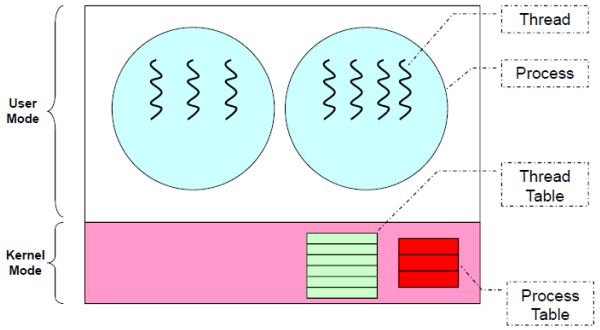
Definition 5.1 (User Thread)**.** *A* ***user thread*** *is implemented as a* ***user library*** *and relevant operations are handled by the runtime system in the process. Kernel will not be aware of the existence of threads in the process. The advantages of user thread model are*



- *Can have multithreaded process on any OS*
- *Thread operations are just library calls*
- *Generally more configurable and flexible, e.g. the scheduling algorithm can be customized.*

However, the disadvantages are

- *OS is not aware of threads, therefore the scheduling of threads across processes is performed at process level. Therefore, one thread blocked will cause the process to be blocked, and therefore all other process will be blocked*
- *It cannot exploit multiple CPUs.*



Definition 5.2 (Kernel Thread)**.**
Thread is implemented in the OS, and thread operation is handled as system calls.
This makes thread-level scheduling possible, and kernel may make use of threads for its own execution. The advantages of kernel thread model are

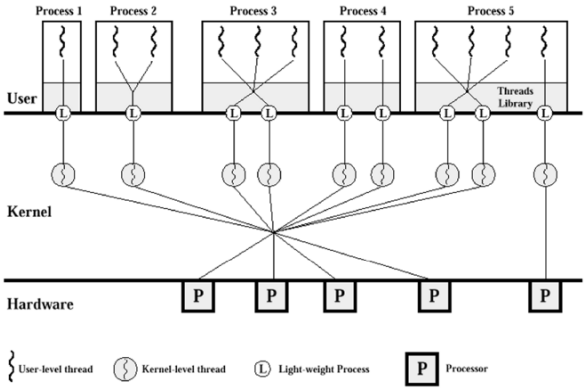
- Kernel can schedule on thread levels, therefore more than 1 thread in the same process can run simultaneously on multiple CPUs.

However, the disadvantages are

- Thread operations is now a system call, which is slower, and more resource intensive
- If implemented with many features, then it will be expensive and an overkill for simple program
- If implemented with few features, it is not flexible enough for some programs

To take advantage of both models, in actual system, we use a hybrid thread model, where

- there are both kernel and user threads
  - OS schedule on kernel threads only
  - User threads are binded to kernel threads



This offers great flexibility, as we can limit the concurrency of any process/user.
Threads on a modern processor starts of a software mechanism and now become hardware native.

5.2 POSIX Threads

The thread in POSIX system can be used by including `#include <pthread.h>`. The useful datatypes are

- `pthread_t`: data type to represent a thread id
- `pthread_attr`: data type to represent attributes of a thread

Creation of a thread is done through `pthread_create(pthread_t* tidCreated, const pthread_attr_t* threadAttributes, void* (*startRoutine) (void*), void* argForStartRoutine`
This call returns 0 for success and non-zero for errors.
Termination of a thread is done through `int pthread_exit(void* exitValue)`
Here, `exitValue` is the value to be returned to whoever synchronize with this thread. If it is not called, a pthread will terminate automatically at the end of startRountine, with no exit value returned.

Thread synchronizatin is done via `int pthread_join(pthread_t threadID, void status)` which waits for the termination of another pthread. It returns 0 for success and nonzero for errors.

6 Inter-Process Communication(IPC)

Since it is hard for cooperating process to share information, we need IPC mechanisms. There are two common IPC mechanisms, namely **shared memory** and **message passing**. There are two *Unix-specific* IPC mechanisms, namely **pipe** and **signal**.

6.1 Shared Memory

Communication via **shared memory** is done via the following steps:

- Process P_1 creates a shared memory region M.
- Process P_2 attaches memory region M to its own memory space.
- P_1 and P_2 can now communicate using this memory region M.

The same model is also applicable to multiple processes haring same memory region.
The advantages of shared memory scheme are

- **Efficient**, since only initial steps of creation and attachment involves PS
- **Easy to use**, since shared memory region behaves the same as normal memory space, so information of any size or type can be written easily.

The disadvantages are

- **Synchronization**: Since it is shared resource, we need to have a proper way for synchronized access
- **Harder Implementation**.

For POSIX Shared Memory sheme, after the steps outlined above,

- Detach M from memory space after use
- Destory M. For this destory operation, only one process needs to do this, and it can only be done if M is not attached to any process

6.2 Message Passing

Communication via **Message Passing** is done via the following steps:

- Process P_1 prepares a message M and send it to process P_2.
- Proecss P_2 receives the message M
- Message sending and receiving are usually provided as system calls

For this model, we need to concern about

- **Naming**: how to identify the other party in the communication

- **Synchronization**: the behaviour of sending/receiving operations

Essentially, since OS is involved in message passing(sending and receiving), the message *have to* be stored in kernel memory space.
There are two naming shemes:

Definition 6.1 (Direct Communication)**.**
Sender/Receiver of the message explicitly name the other party. For example, `Send(P2, Msg); Receive(P1, Msg)`.
The characteristics of this scheme are

- We need one link per pair of communicating processes
- We need to know the identity of the other party

Definition 6.2 (Indirect Communication)**.**
Message are sent to/received from message storage, usually known as **mailbox** or **port**. All messages are sent or retrieved from mailboxes.
The characteristics of this scheme is that one mailbox can be shared among a number of processes.

There are also two synchronization behaviours:

Definition 6.3 (Blocking Primitives(Synchronous))**.**

- `Send()` will cause sender to be blocked until the message is received.
- `Receive()` will cause receiver to be blocked until a message arrives.

Definition 6.4 (Non-Blocking Primitives(Asynchronous))**.**

- `Send()` will have sender resume operation immediately.
- On `Receive()`, receiver either receive the message if available, or some indication that message is not ready yet.

The advantages of message passing is that

- **Portable**: can easily be implemented on different processing environment
- **Easier Synchronization**: especially synchronous primitive is used, sender and receiver are implicitly synchronized.

The disadvantages are

- **Inefficient**, as it requires OS intervention
- **Harder to use**, as message are usually limited in size and/or format

6.3 UNIX Pipes

In Unix, a proces has 3 default comminucation channels:

- stdin
- stderr
- stdout

Unix shell provides the "|" symbol to link the input/output channels of one process to another, and this is known as **piping**.
Essentially, a pipe can be shared between two processes, which makes the two processes to form a Producer-Consumer relationship.
The pipe behaves like an anonymous file, and access of this pipe is FIFO, i.e. access of data must be in order.

In implementation, pipe functions are implemented as **circular bounded byte buffer** with implicit synchronization:

- Writer will wait when buffer is full
- Reader will wait when buffer is empty

Remark: In variants of pipe, we can have multiple readers and writers. Also, the pipe can be either **half-duplex** or **full-duplex**.

The `C` system call is `int pipe(int fd[])`, where `fd[0]` is the reading end and `fd[1]` the writing end.

If we wan tto change the standard communication channels, we can use system calls named `dup()` and `dup2()`.

6.4 Unix Signal

Unix Signal is a form of inter-process communication, which serves as an asynchronous notification regarding an event sent to a process or a thread.
The recipient of the signal must handle the signal, by

- A default set of handlers, or
- User supplied handler(only applicable to some signals)

The common signals in UNIX include kill, stop, continue, memory error, arithmetic error etc.

7 Synchronization

The lack of synchronization will cause problem with **shared, modifiable resources** when concurrent processes access it in a interleaved fashion. This is known as **race condition**. To resolve this, we need to implement a **critical section** where only one process can enter at any time. Correct Critical Section requires the following properties to be satisfied:

1. **Mutual Exclusion**: If process P_1 is executing in critical section, all other processes are prevented from entering the critical section
2. **Progress**: If no progress is in a critical section, one of the waiting processes should be granted access.
3. **Bounded Wait**: After process P_1 request to enter critical section, there exists an upper bound of number of times other process can enter the critical section before P_1.
4. **Independence**: Process *not* executing in critical section should never block other process.

In case of incorrect synchronization, there will be problems like

1. Deadlock, where all processes are blocked
2. Livelock, where processes keep changing state to avoid deadlock and make no other progress
3. Starvation, where some processes are blocked forever

7.1 Assembly Level Implementation

We require an **atomic** instruction, named `TestAndSet`:

```
TestAndSet Register, MemoryLocation
```

The behaviour of the `TestAndSet` is to

1. Load the current content at `MemoryLocation` into `Register`
2. Stores a 1 into `MemoryLocation`
3. Returns the Register's value to the user

Therefore, `TestAndSet` will return 1 if and only if the resource is used by another process. With this observation, we can code `EnterCS` and `ExitCS` as follow:

```
void EnterCS(int* lock){
    while(TestAndSet(lock)==1);
}
void ExitCS(int* lock){
    *lock = 0;
}
```

The implementation satisfies all four requirement of critical section, but the drawback is also evident: it employs a **busy waiting** scheme when waiting to enter CS, which is a wasteful use of processing power. We hope that such wait can essentially be turned into a blocked state of the process.

7.2 High Level Language Implementation

Theorem 7.1 (Peterson's Algorithm)**.**
Suppose there are two processes P_0 and P_1. We will use

1. `Turn`, a variable that is set to indicate the turn belongs to the other process before checking for whether it is permitted to enter critical section
2. `Want` array of 2 elements. This `Want` array allows process to indicate the intention of entering the critical section.

Suppose process 0 wants to enter critical section, it will set `Want[0]=1` and `Turn=1`, and check whether `Want[1]&&Turn==1` is true, which means that opponent wants and has the turn to use the critical section. If it is true, it will use a while loop to block P_0's entrance. After it successfully finishes critical section, `Want[0]=0`.

The assumption here is **writing to turn** is **atomic**.
The drawback of this algorithm is

- Buzy Waiting
- Low level
- Not general, as we may require more than mutual exclusion for synchronization

7.3 High Level Synchronization Mechanism

Definition 7.1 (Semaphore)**.**
A semaphore is a generalized synchronization mechanism, which provides

- a way to block a number of processes, known as **sleeping process**, and
- a way to unblock/wake up one, or more sleeping process

Semaphore will contain a integer valued number S. This S can be initialized to any non-negative values initially.
Semaphore supports two **atomic** semaphore operations: `Wait(S)` and `Signal(S)`.

```
Wait(S)
    if(S<=0) blocks
    decrement S
Signal(S)
    increment S
    wake up one sleeping process if any
```

The invariant of semaphore is

$$S_{\text{current}} = S_{\text{initial}} + \#\text{signal}(S) - \#\text{wait}(S)$$

where $\#\text{signal}(S)$ is number of signals operations executed, and $\#\text{wait}(S)$ is the number of wait operation **completed**.
With this invariance, we have $S_{\text{current}} + N_{\text{CS}} = 1$ for binary semaphore. This ensures there is no deadlock.

7.4 Classical Synchronization Problems

7.4.1 Producer Consumer

Suppose there is a shared bounded buffer of size K, where producer can produce item into the buffer if buffer is not full, and consumers can remove item from the buffer if buffer is not empty. There are multiple consumers and multiple producers. is a blocking solution, if we

```
while (TRUE) {
        Produce Item;

        wait( notFull );
        wait( mutex );
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        signal( mutex );
        signal( notEmpty );
}
```

Producer Process

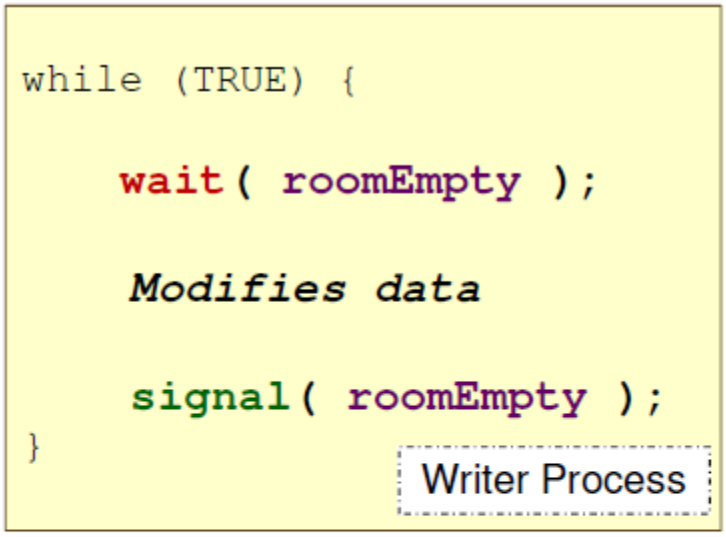
```
while (TRUE) {

        wait( notEmpty );
        wait( mutex );
        item = buffer[out];
        out = (out+1) % K;
        count--;
        signal( mutex );
        signal( notFull );

        Consume Item;
}
```

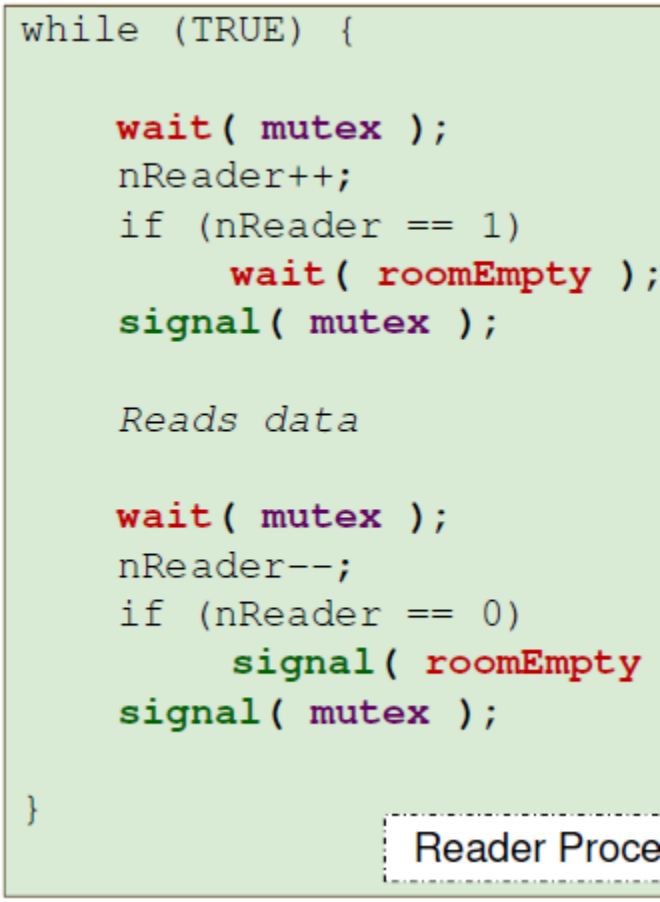
Consumer Proce

initialize count, in , out to be 0 and mutex has $S = 1$, notFull has $S = K$ and notEmpty has $S = 0$.

7.4.2 Reader Writer

Suppose there is a data structure D where reader can retrieves information from D and writer can modify information from D. Here we grant writer **exclusive access** to D, whereas multiple readers can access at the same time. is a solution that favours reader.





- Initial Values:
  - roomEmpty = S(1)
  - mutex = S(1)
  - nReader = 0

7.4.3 Dining Philosophers

There are 5 philosophers seating around the table, either thinking, hungry or eating. There is one chopstick to the left and another to the right of each philosopher, in total 5. We aim to let some of the philosopher to dine by successfully taking up 2 chopsticks that is on its left and right.
The code is detailed in the lecture notes.
An alternative is to use limited eater, where will use a semaphore with $S = 4$ to limit number of seats to 4. and we use a semaphore on each chopsticks.

8 Memory Abstraction

Memory is an array of bytes, with a unique index for each byte, known as physical address. For process, there are 4 regions: text, data, heap and stack. Out of these 4, how text and data are layed outis decided by compiler in executable.
There are two types of data in a process:

- **Transient** data, which is valid only for a limited duration, e.g. function parameter, local variable
- **Persistent** data, which is valid for duration of program unless explicitly removed, e.g. global variable, constant variable, dynamically allocated memory

Note: both types of data section can grow/shrink during execution.

8.1 Memory Abstraction

Without memory abstraction, memory access is straightforward, as address is fixed during compilation time, and such address is the physical address. However, different process may face conflict and it is also hard to protect memory space.
One simple abstraction is the usage of base and limit registers.

Definition 8.1 (Base and Limit Registers)**.**
Use a special register as base of all memory references, which is known as **base register**. During compilation time, all memory references are compiled as *offset* from this register. At loading time the base register is initialized as the starting address of the process memory space.
Use another special register to indicate the range of the memory space of the current process, which is known as **limit register**. All memory access is checked against the limit to protect memory space integrity.

The problem with this approach is that to access any address, we need to do more computations:

- `Actual` = `Base` + `Adr` to get actual address.
- Check `Actual` < `Limit` for validity.

The idea evolves to be the idea of **logical address**, which has a bijection to physical address. Building on this abstraction, each process will now have a self-contained, independent logical memory space.

8.2 Contiguous Memory Management

In this subsection, we assume

1. Each process occupies a **contiguous memory region**.

2. The **physical memory is large enough** to contain one or more processes with complete memory space.

Here, to support multitasking, we allow multiple process to be in the physical memory at the same time for efficient switching. When physical memory is full, we free up memory by

- removing terminated process, or
- swapping blocked process to secondary storage.

Definition 8.2 (Memory Partition)**.**
Memory Partition is the contiguous memory region allocated to a single process.

We can make these partitions in two ways, either **fixed-sized partition** or **variable-sized partition**.

Definition 8.3 (Fixed Sized Partition)**.**
For fixed sized partition, physical memory is split into fixed number of equally sized partition and a process will occupy one of them.
It has the advantage of easy management and fast allocation. However, partition size need to be large enough to contain the largest of the processes.
Therefore, smaller process will waste memory space, known as **internal fragmentation**.

Definition 8.4 (Dynamic Partitioning)**.**
For variable-sized partition, partition is created based on actual size of process. OS keep track of the state of occupied and free memory regions and perform splitting and merging when necessary.
It is flexible and avoid internal fragmentation, but OS needs to maintain more information, and it takes more time to locate appropriate region. Also, **external fragmentation** occurs, which creates a large number of holes.

The allocation algorithm for dynamic partitioning can be

- First fit: take the first large-enough hole
- Best fit: find the smallest hole that is large-enough
- Worst fit: find the largest hole

After finding the hole, split.
When occupied partition is freed, algorithm will merge it with adjacenet hole if available.
Suppose the holes are fragmented, **compaction** can be used to move the occupied partition around to create consolidate holes.
The OS usually stores the memory information into a linked list of 3-tuple: (`Status`, `Start Address`, `Length`).

Theorem 8.1 (Buddy System)**.**
Buddy memory allocation provides efficient

- Partition Splitting
- Locating good match of free partition
- Partition de-allocation and coalescing

Suppose memory is of 2^K byte, we keep an array of `A[0..K]`. Each array element `A[j]` is a linked list which keep tracks of **free** blocks of size 2^j. Each free block is indicated by the starting address.
To allocate a block of size N,

1. Find smallest S such that $2^S \geq N$.
2. Access `A[S]` for a free block.
   (a) If free block exists, remove block from `A[S]` and allocate
   (b) Else, find the smallest `R` from `S+1` to `K`, such that `A[R]` has a free block B. Repeatedly split B until there are free blocks in `A[S]` and allocate.

To deallocate a block B,

1. Check in `A[S]`, for the buddy of B, say C. If it exists, remove B and C, and merge to B', and try deallocate B'.
2. Else, insert B in `A[S]`.

Note, two blocks are buddy of size 2^S, if the S bit of B and C is a complement and leading bits up to Sth bit of B and C are the same.

9 Disjoint Memory Schemes

In this section, process memory space can now be in **disjoint physical memory locations**. This can be done via paging or segmentation.

9.1 Paging

The **physical memory** is split into regoins of fixed size, known as **physical frame**. The **logical memory**, which has same size as physical, is also split into regions of *same size*, known as **logical page**.
At execution time, pages of a process are loaded into *any available* memory frame. The process will still occupy a contiguous logical memory space.
Under paging scheme, we will keep a mapping of logical page to corresponding physical frame using **page table**. Essentiallly, the logical memory of a process always start with page 0, so the page table `Table` can be an 0-indexed array.
Suppose the physical frame size, which is also page size, equals S, then the physical address of kth byte of ith page equals $\texttt{Table}[i] \times S + k$.
In practice, we

- Keep frame size and page size as power-of-2
- Also, physical frame size equals logical page size

Suppose we have 2^n bytes in a page, then the logical address can be represented by $\underbrace{p}_{(m-n)}\underbrace{d}_{n}$.
The corresponding physical address is $\underbrace{f}_{(m-n)}\underbrace{d}_{n}$, where $f = \texttt{Table}[p]$.

Paging removes **external fragmentation**, but there exists, still, **internal fragmentation**.

To implement efficient paging scheme, we need **translation look-aside buffer**(TLB). TLB is a cache of a *few* page table entries.
TLB works as below:

- Page number is used to search TLB associatively (in parallel)
- If TLB hit, then frame number is retrieved for translation
- If TLB miss, memory access is done to access the full page table, update TLB and do translation

When a context swtich occurs, TLB entries are flushed. When process is switched back, TLB miss will occur to fill TLB. It is still possible to place some entries initially, like the code pages, to reduce TLB misses.

Paging scheme can provide protection of memory using access-right bits and valid bit.
Access Right bit is attached to each page table entries to indicate whether it is writable, readable or executable. Memory access is checked against the access right bits.

Valid bit is attached to each page table entry to indicate whether the page is valid for process to access. This bit is set by OS and out-of-range will be caught.
The page table allows several processes to share the same physical frame, by putting the same physical frame number in page table entries. Therefore, it can be used to

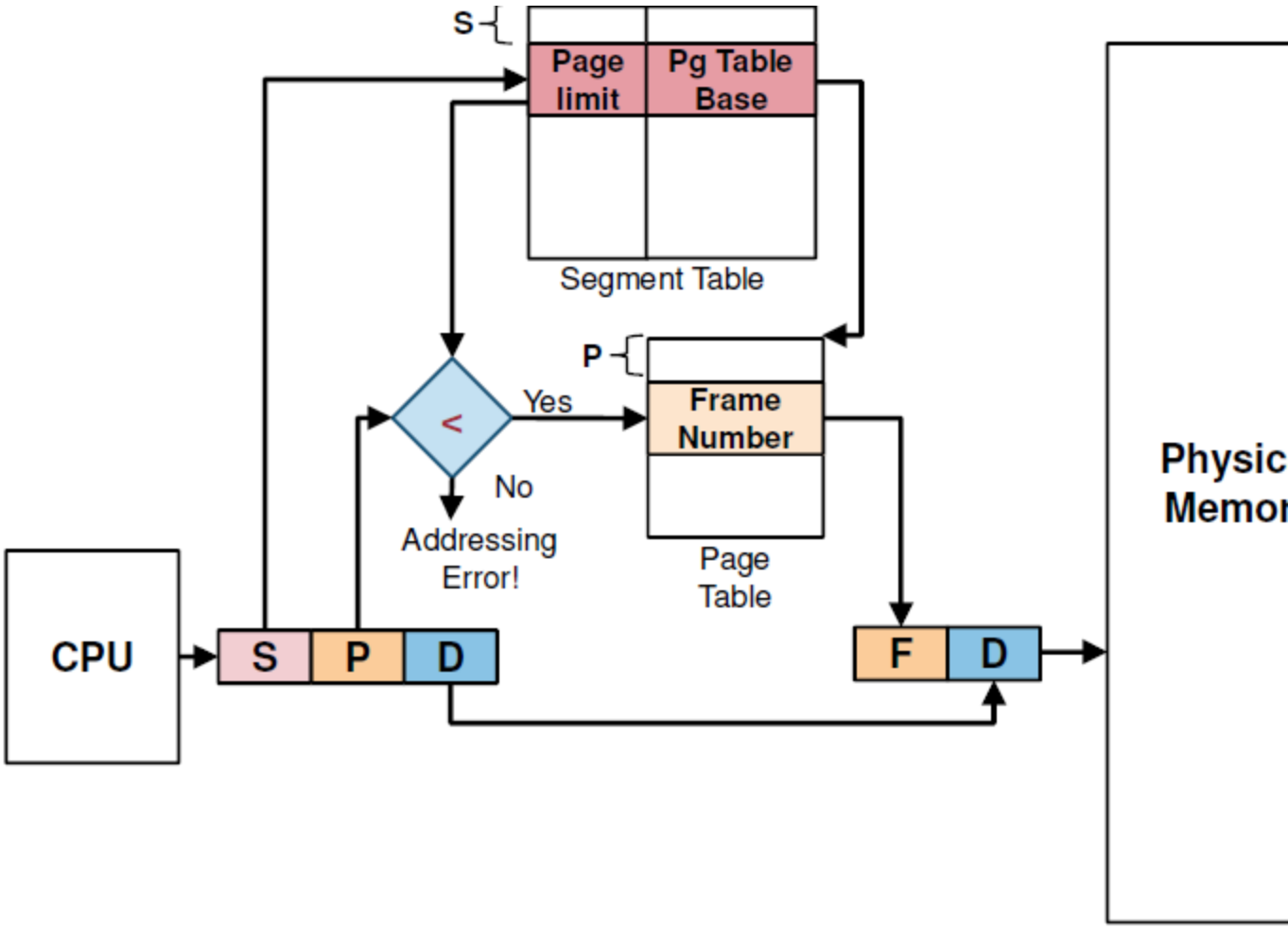
- Share code page
- Implement copy-on-write

9.2 Segmentation Scheme

It is hard to place different regions(data, text, stack, heap) in contiguous memory space, physical or logical, and allow growing/shrinking freely. Therefore, we use **segmentation scheme**, which separate regions into multiple memory segments. Logical memory space is then a collection of segments.
Each memory segment will have a name, and have a limit to indicate the range. The memory reference is now specified as `segment name + offset`.
Each process will keep track of a **segment table**, whjich is a mapping between different segments, identified by `segID` to a pair: `<Base, Limit>`. Suppose the logical address is `SegID` with offset k, then the physical address is `Table[SegID]+k`, if k is less than limit.
Since we need to check limit and also do addition for offset, we do not hardware to support these operations.
Segmentation scheme is good since each segment can grow/shrink independently and can be protected/shared independently.
However, segmentatino requires variable-size contiguous memory regions, which can cause external fragmentation.

9.3 Segmentation with Paging

The idea is to page each segment.

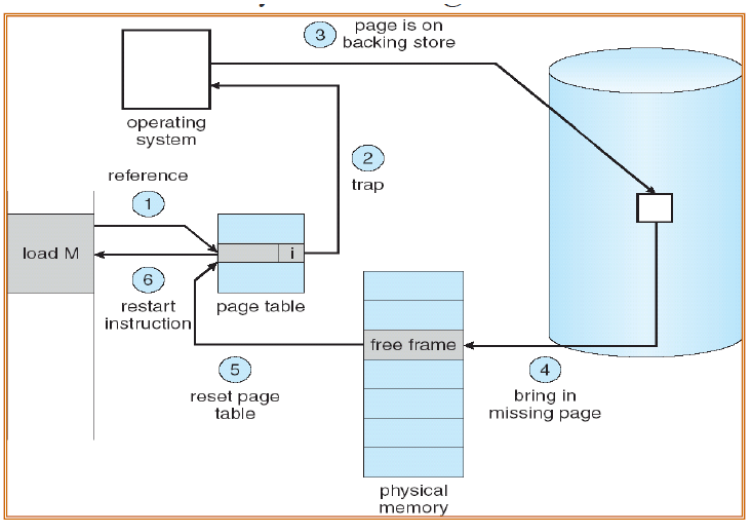

S
Page limit
Pg Table Base
Segment Table
P
Frame Number
Page Table
<
Yes
No
Addressing Error!
CPU
S
P
D
F
D
Physic
Memor

10 Virtual Memory Management

In virtual memory management, we allow logical memory space of a process tobe larger than physical memory. The logical address space can either reside in physical memory or in *secondary* storage.
We extend the paging scheme:

- Logical memory space is split into fixed size page
- Some pages may be in physical memory
- Others in secondary storage

We use a **page table** to translate virtual address to physical address. In the page table, we add an additional field to distinguish between a **memory resident** and non-memory resident.
If CPU finds out a certain page is *not* memory resident, it will trigger a **page fault**, which will be handled by OS, to bring the non-memory resident page into the physical memory. After this step, CPU will retry the instruction. It is usually unlikely that **thrashing**, the



occurrence of page fault, will happen, because of **locality principle**, which includes temporal and spatial locality.

- Temporal Locality: after a page is loaded to physical memory, it is likely to be accessed in near future, so that cost of loading page is amortized.
- Spatial Locality: a page contains contiguous locations that is likely to be accessed in near future, so later access to nearby locations will not cause page fault.

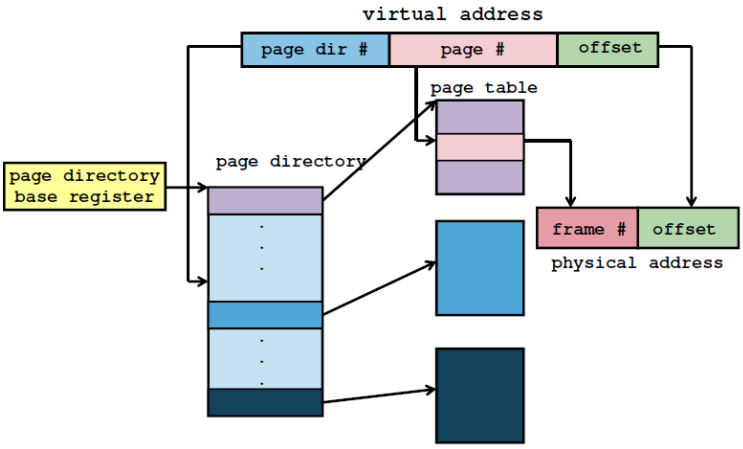
Virtual memory completely separate logical memory address from physical memory, as amount of physical memory no longer restrict the size of logical memory address.

It is a more efficient use of physical memory, as page currently not needed can be on secondary storage.
It allows, also, more proecsses to reside in memory, so that more processes can be chosen to run by CPU, increasing CPU utilization.

10.1 Page Table Structure

In case of huge memory space, page table via **direct paging** will also be huge, and remain fragmented in different pages. Therefore, we need a neater design of page table.

Definition 10.1 (2-Level Paging)**.**
We split the full page table into smaller page tables, each with a **page table number**. If the original page table has 2^P entries, with 2^M smaller page tables, M bits is needed to unique identify one page table; each smaller page table contains 2^{P-M} entries.
Each smaller page table should ideally be of the same size as the page size, to avoid page fault. To keep track of these smaller page tables, a single **page directory** is needed, which contains the 2^M indices to locate each of the smaller page table.

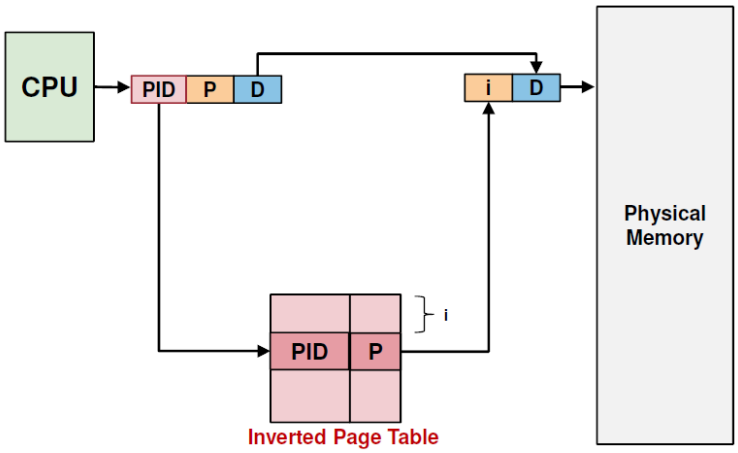


It allows empty entries in the page directory, which means the corresponding smaller page table need not be allocated. This saves memory space.

Definition 10.2 (Inverted Page Table)**.**
Normal page table is per-process information. Also, mapping from physical frame to logical memory is inaccessible.
Inverted Page Table keeps a **single** mapping of physical frame to $\langle \text{pid}, \text{page number} \rangle$. Here, the page number is specific to the pid.
The offset of the inverted page table entry will be the frame number of physical frames.
To make query faster, one should hash the inverted page table.

10.2 Page Replacement Algorithm

In the case of page fault, we need to evict(free) a memory page, by writing back the old page to the memory if the page id modified, therefore **dirty**.



To decide which page to evict, we need to look at a suitable replacement algorithm. A good replacement algorithm should **reduce the total number of page faults**.

Definition 10.3 (Optimum)**.**
Optimum page replacement algorithm replace the page that **will not be used again for the longest period of time**.

- If the page is in memory, only update **next use time**.
- Else,
  - If memory is not full, load it and update **next use time**.
  - Else, choose the existing page with largest next use time to evict, and load the new page.

This guarantees minimum number of page faults.
However, it is not realizable, due to requirement of future knowledge of memory access.

Definition 10.4 (FIFO Page Replacement)**.**
Memory pages are evicted based on their loading time. One evicts the oldest memory page. Therefore,

If the page is in memory, use.

Else,

- If memory is not full, load it and update **loaded-at time**.
- Else, choose the existing page with smallest loaded-at time to evict, and load the new page.

The problem with FIFO algorithm is Belay's Anomaly, where the number of page fault can increase if the number of physical frame increases. The reason is FIFO does not exploit temporal locality.

Definition 10.5 (Least Recently Used(LRU))**.**
In LRU, we replace the page that has not been used in the longest time. It avoids Belady's Anomaly.

If the page is in memory, only update **last use time**.

Else,

- If memory is not full, load it and update **last use time**.
- Else, choose the existing page with smallest last use time to evict, and load the new page.

However, implementing LRU is not easy, because the last use time can overflow, and also the searching of smallest last use time page requires a complete search.

Definition 10.6 (Second-Change Page Replacement)**.**
It used a modified FIFO to give a second chanmge to pages that are accessed. We need the page table entry to maintain a reference bit:

- 1 = Accessed
- 0 = Not Accessed.
- Initialize the next victim pointer to the first frame when it is loaded.
- If the page is in memory, set reference bit to 1.
- Else,
  - If memory is not full, load it and set reference bit to 0.
  - Else, we look at the next victim pointer.
    - If next victim pointer points to a page with reference bit 0, replace that page.
    - Else, advance victim pointer and check again.
    - If all pages have reference bit 1, change the original victim to the new page.

10.3 Frame Allocation

To allocate N physical memory frames among M processes, some simple appoaches can be

- Equal distribution
- Proportional allocation to the size of process.

During page replacement, if victim page is selected among **page of the same process** that causes page fault, it is known as **local replacement**. Otherwise, it is known as **global replacement**.
Local Replacement has advantage of stable performance between multiple runs, since number of pages allocated remains constant.

However, it has disadvantage of potentially hindering the progress of a process if page is not allocated enough.
Global Replacement has the advantage of allowing self-adjustment between processes, so that process that needs more frame can get from others.
However, badly behaving process can affect others.
If global replacement is used,

- Thrashing steals page from other process, which can cause cascading thrashing

If local replacement is used,

- Although thrashing is localized, single process doing thrashing can hog the I/O and degrades performance of other processes.

Definition 10.7 (Working Set Model)**.**
We define the working set Window δ as a interval of time. We denote $W(t, \delta)$ to be the active pages in interval at time t. We allocate enough frames for pages in $W(t, \delta)$ to reduce possibility of page fault.
The accurancy of working set model is directly affected by the δ chosen.

11 File Management Introduction

We want to introduce a file system on top of external storage to store **persistent** information, in contrast to physical memory storing **volatile** information.
File system sdhould provide

- Abstraction on top of physical media
- A high level resource management scheme
- Protection between processes and users
- Sharing between processes and users

The requirement of file system needs to be

- **self-contained**, where information stored on a media is enough to describe the entire organization; should be able to "plug and play" on another system
- **Persistent**: data should persist beyond the lifetime of OS and processes
- **Efficient**: it should provide good management of free and used space, and there should be minimum overhead for bookkeeping information

11.1 File System Abstractions

File System consists of a collection of **files** and **directory** structures, where

- **File**: an abstract storage of data, and
- **Directory**: Organization of files

File systems provides an abstraction of accessing and using the files and directories.
File represents a logical unit of information created by process. It can be viewed as an abstract data type, in which there are

- **Data**: Information structured in some ways
- **Metadata**: Additional Information associated with the file, also known as file attributes. Common metadata fields include
  - Name: a human readable reference to the file.
    Different file system will have different **naming rule**. Some system include file extension in file name like `Name.Extension` so as to use the extension to indicate **file type**.
  - Identifier: A unique id for the file used internally by File System
  - Type: indicate different type of files.
    There are 3 common file types:

    * Regular files: contains user information. Inside regular files, there are two major types:
      · ASCII files, which can be displayed as it is
      · Binary files, which include executable, PDF files etc. It has a **predefined internal structure** that can be processed by specific programme.

      We can use either **file extension**(windows) or **embedded information**(unix) to distinguish file types.
    * Directories: system files for File System Structure
    * Special files: character/block oriented
  - Size: current size of file
  - Protection: Access permissions, can be classified as reading, writing and execution rights
    We can specify the type of access certain user or group of user can do on the file. The type of access include:
    * Read: retrieve information from file
    * Write: Write/rewrite the file
    * Execute: Load file into memory and execute it
    * Append: Add new information to the end of the file
    * Delete: Remove file from File System
    * List: Read metadata of the file

    The most general scheme to specify these rights is to use a **access control list**, in the format of a list of user identity with their allowed access types. It is very customizabole but additional information needs to be associated with the file.
    Another way to protect the file is to use permission bits, covered in `CS2107.pdf`.
  - Time, date and owner information: creation, last modification time, owner id etc
  - Table of content: Infomration for the file system to determine how to access the file

and various operations can be done on the file.
Operations that can be done on file metadata include:

- Rename
- Change attributes, like file access permissions, dates, ownership etc
- Read attribute, like file creation time

File data can have three different ways for abstraction:

- **Array of bytes**: there is no interpretatino of data, so the data are just raw bytes. Each byte has a unique offset from the file start.
- **Fixed length records**: array of records, which can grow/shrink. It can jump to any record easily since the offset of nth record is size of record$\times(n-1)$.

- **Variable length records**: which is flexible but harder to locate a record

There are also 3 kinds of access methods:

- Sequential Access, which reads data in order, starting from beginning. Data cannot be skipped but can be rewound.
- Random Access, where data can be read in any order. This can be realized using
  - `read(offset)`, which explicitly state the position to be accessed
  - `seek(offset)`, which moves from old to a new location in the file. (used by windows and unix)
- Direct Access: which is used for files that contains fixed-length records. It allow **random access to any record directly**.
  Basic random access method is a special case of direct access, where each record equals 1 byte.

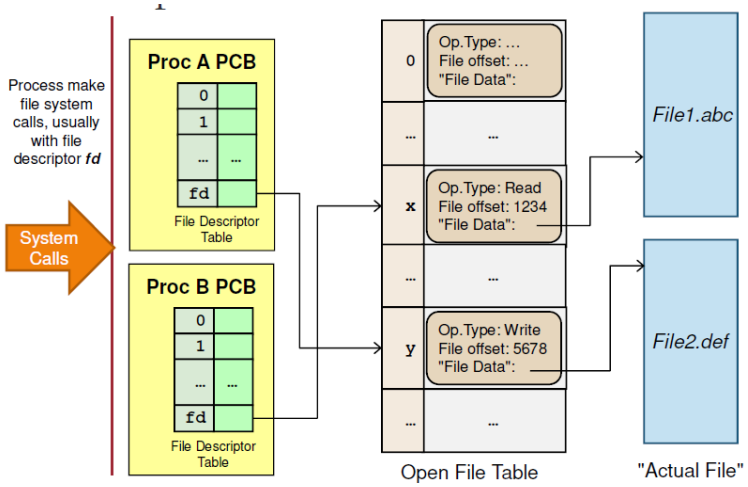
There are a few operations that can be done on file data:

- Create: create a new file with no data
- Open: perform before further operations, so as to prepare necessary information for file operations later
- Read: read data from file, usually starting from current position
- Write: write data to file, usually starting from current position
- Repositioning: which is seek.
- Truncate: removes data between specified position to the end of file

These operations are provided by OS via **system calls**, which provides protection, concurrent and efficient access, and also maintain necessary information. Especially, OS need to keep information for an **opened** file in a open-file table:

- File Pointer: current location in file
- Disk Location: actual file location on disk
- Open Count: How many process has the file opened.

To keep track of these information, we need a system-wide open-file table and also some per-process open-file table. The system-wide table's entry corresponds to a unique file, whereas the per-process open-file table entry corresponds to a entry in a system-wide table.



11.2 Directory

Directory is used to

- Provide a logical grouping of files
- Keep track of files

We can structure the directory as

- Single-level
- Tree structure, which allows us to have an **absolute pathname**, and a **relative pathname** from the **current working directory**.
- DAG, by allowing files to be shared. This sharing can be done using
  - Hard Link, which can only be used on files.
    A hard link of a file F in directory A to another directory B makes A and B has separate pointers pointing to the actual file in disk. This has low overhead, since only pointers are added in directory, but deleting the file may cause problems. Hard link can be done using `ln`.
  - Symbolic Link, which can be used on files and directory.
    In the same scenario above, now B will create a **special link file** G where G contains the path name of F.When G is accessed, it finds out where is F and access F.
    This solves the problem of deletion, but has larger overhead. To achieve symbolic link, use `ln -s`.
- General Graph
  This is not desirable, since we may encounter infinite loop in traversal and it is hard to determine when to remove a file or directory.

12 File System Implementation

General disk structure can be treated as a 1D array of **logical block**, where the logical block is the smallest accessible unit.
Logical block is mapped into **disk sectors**, whose layout is **hardware dependent**.
The **Master Boot Record(MBR)** at sector 0 with partition table, is followed by one or more partitions, where we put an independent file system in each partition.
A file system generally contains:

- OS bootup infomration
- Partition details, e.g., total number of blocks and number and location of free disk blocks
- Directory structure
- File information
- Actual file data

12.1 Implementing File

File is a collection of logical blocks. Therefore, there could be internal fragmentation if the file size is not a integer multiple of logical blocks.
A good file implementation must

- Keep track of logical blocks
- Allow efficient access
- Disk space is utilized effectively

12.1.1 Contiguous Block Allocation

In contiguous block allocation, we allocate consecutive disk blocks to a file.
It is simple to keep track of the file since we only need

- starting block number
- length

However, there will be external fragmentation, since we can leave holes as files get deleted.
Also, for this scheme, file size needs to e specified in advance.

12.1.2 Linked List Allocation

We keep a linked list of disk blocks, where each block stores:

- The next disk block number
- Actual file data

The file information will store first and last disk block number.
Linked list implementation solves the external fragmentation, however,

- Random access is very slow
- Part of the disk block is used for pointer
- Less reliable, since each pointer value can be a single point of failure

12.1.3 Linked List v2.0

Move all the block pointer into a single table, known as **File Allocation Table(FAT)**. The FAT is in *memory* all the time.
This allows faster random access, since linkedlist traversal takes place in memory.
However, FAT keeps tracks of all disk block in a partition, so it can be huge when disk is large, and will consume valuable memory space. **Remark**: For the Linked List Allocation, the last block is *NULL*.

12.1.4 Indexed Allocation

In indexed allocation, each file has an **index block**, which is an array of disk block addresses, where `indexBlock[N]` is the Nth block address.
The advantage is that we have lesser memory overhad, since only the indexed block of opened file needs to be in memory. This also allow fast direct access.
However, we have a limited maximum file size, since max number of blocks needs to be no larger than the number of index block entries. Also, there is a index block overhead.

12.1.5 Indexed Allocation Variant

We can use a linked scheme to keep a linkedlist of index block, where each index block contains the pointer to next index block.
We can also use multilevel index, where the higher level index block points to a number of lower level index blocks, and the leaf level index block point to actual disk blocks.

12.2 Free Space Management

We need to know the location of free space since we need these information for file allocation. Therefore, we should maintain a **free space list**, and support two functionalities:

- Allocate, during free creation or enlargement
- Free, during file deletion and truncation

12.2.1 Bitmap

Each disk block can be represented by 1 bit in the bitmap, where 1 means **free** and 0 means **occupied**.
Bitset is good since it provides a good set of manipulations, however, we need to keep in memory the whole bitmap for efficiency reasons.

12.2.2 LinkedList

We can use a linked list of disk blocks, where each disk block contains a number of free disk block numbers and a pointer to the next free space disk block. The linked list is stored in the free blocks themselves.

12.3 Implementing Directory

We need to

- keep track of files and subdirectories in the directory, possibly with the file metadata
- Map the file name to the file information

Given a full path name, we need to recursively search the directories along the path to arrive at the file information.
We can view subdirectory as a file entry with special type in the directory.

12.3.1 Linear List Implementation

Directory consists of a list, where each entry represents a file. The entry should contain

- file name(minimum) and possibly other metadata
- file information or pointer to file information

Locating a file using list requires a linear search, which will be inefficient for large directories or deep tree traversal. To resolve this, we can use cache to remember the latest few searches.

12.3.2 Hash Table

Each directory contains a hash table of size N. When we locate a file by file name, we can use the hash.
It has fast lookup speed but hash table has limited size, and also the performance depends on good hash functions.
For file information, it should consists of

- File name and otehr metadata
- Disk blocks information

We can

- Store everything in directory entry, or
- Store only file name and points to some data structure for other information

12.4 File System in Action

Previously, we concern about static information for a file system. However, at runtime, when user interacts with file, runtime information is needed, which should be maintained by OS in memory, e.g., system-wide and per-process open-file table, and buffer.

12.4.1 FIle Creation

We need directory structure to locate the parent directory. Search for filename to avoid duplicate. THe searching can be on cached directory structure.
Then we use free space list to find free disk blocks.
The we add an entry to parent directory, with relevant file information such as file name and disk block information.

12.4.2 File Open

1. We first seafch system-wide table for existing entry. If found, create an entry in P's table pointing to E and return a pointer to this entry. If not, continue to next step.
2. Use full pathname to lcoate file. If not found, terminate with error.
3. When F is located, its file infomration is loaded into a new entry E in system-wide table
4. Create an entry in P's table to point to E, and return a pointer to this entry

The returned pointer is used for further read/write operation.

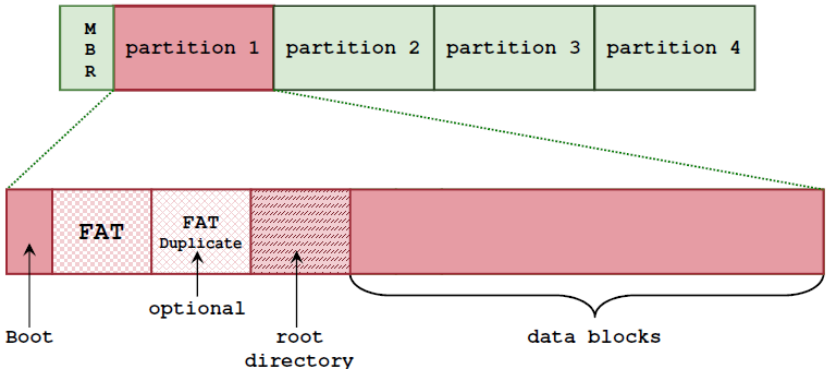
12.5 Disk I/O Scheduling

The time taken to perform a read/write operation= Seek time + Rotational Latency + Transfer Time.
The tiem taken to position the disk head over the proper track is known as **seek time**.
The time to wait for the desired sector to rotate under the read/write head is known as **rotational latency**.
The time taken to transfer the data is known as transfer time, which is $\frac{\text{Transfer size}}{\text{Transfer Rate}}$.
Usually, first two time are more significant than the third. Therefore it is worthwhile for OS to do scheduling. A few algorithms include

- FCFS
- Shortest Seek First
- SCAN family, which can be bidirectional or unidirectional.

13 File System Case Study

13.1 FAT File System



Above, the FAT is called File Allocation Table.
File data are allocated to a number of data blocks, or data block cluster. The allocation info is kept as **linked list** and stored separately in the FAT.
In the FAT, there will be one entry per data-block/cluster. It stroes the block information such as whether theblock is free, next block, damaged etc.(see below)
OS will cache the FAT into the RAM to facilitate linked list traversal.
FAT entry contains either

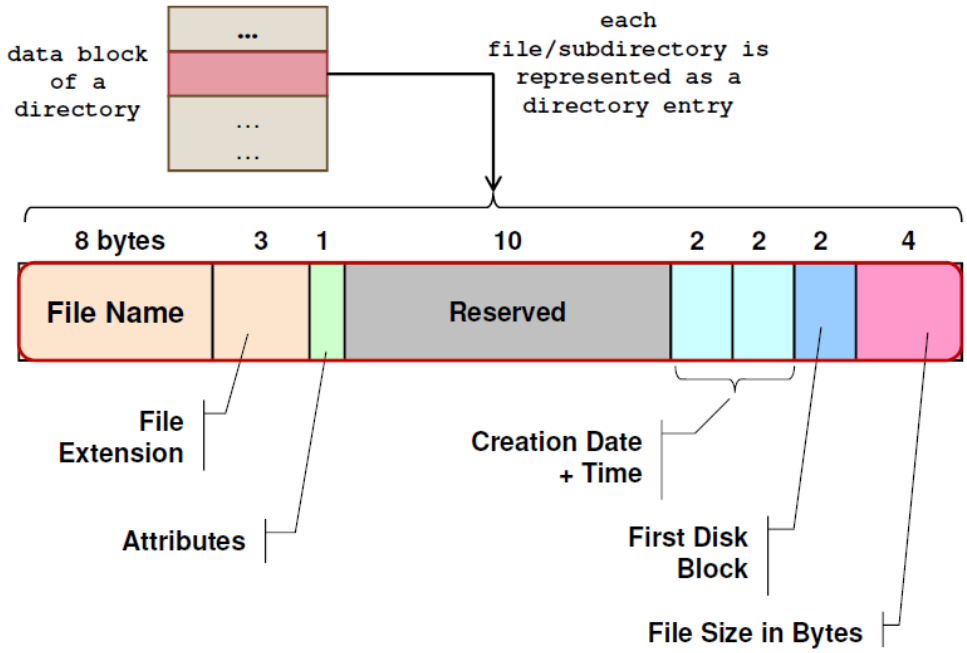
- FREE block
- Block Number of next book
- EOF code(i.e. NULL pointer)
- BAD block

Directory, on the other hand, is represented as a

- **special type of file**
- Root directory is stored in a *special* location, whereas other directories are stored in data blocks
- Each file/subdirectory within the directory is represented as **directory entry**

Directory entry is a fixed-size 32-bytes per entry, which contains

- Name, Extension
- Attributes
- Creation Date, Time
- First disk block, file size

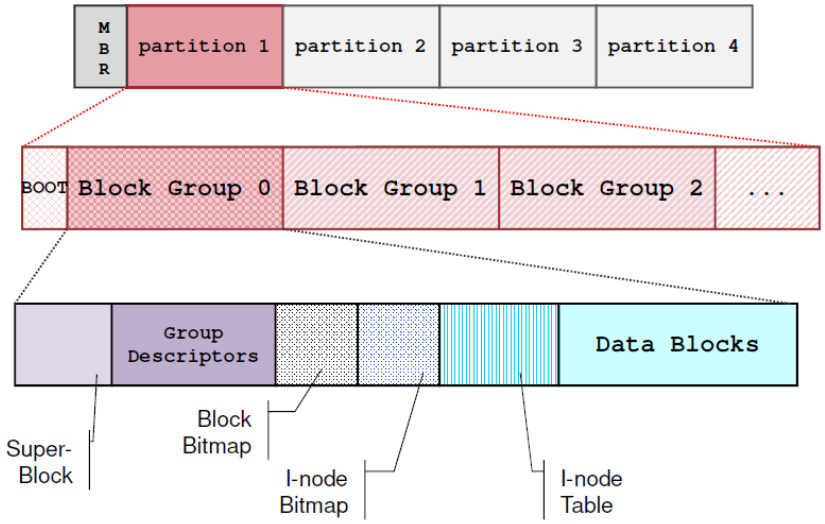


13.1.1 File Operations

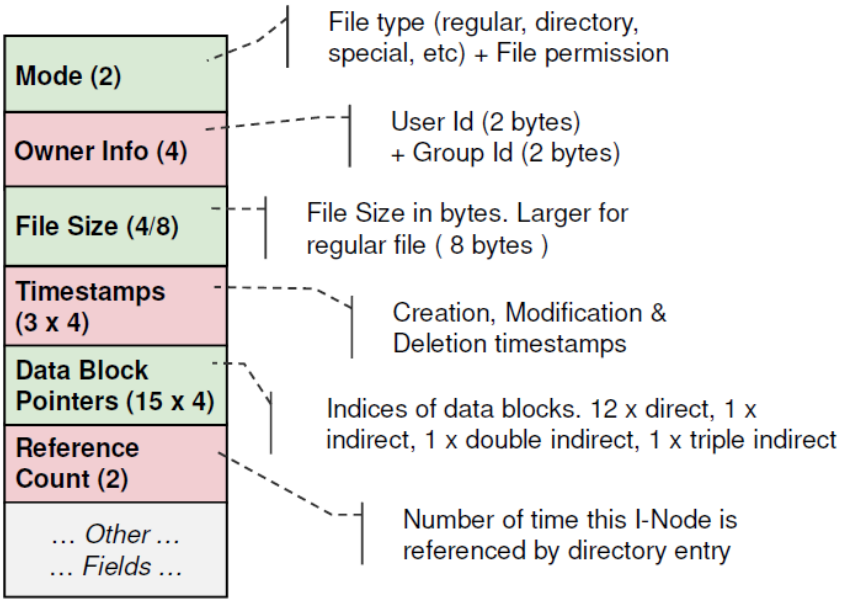
For **File Deletion**, we *delete* teh directory entry by settting the first letter in the filename to the special value `0xE5`. We then free data blocks by setting FAT entries to FREE. However, actual data blocks remain intact.
However, we do not record the free spaces, so we must calculate it via traversal of FAT table. Instead of pointing to the disk blocks, FAT indexes teh disk cluster. FAT32 allows 2^28 clusters, subtracted by the special values for EOF etc. Cluster size will affect condition of internal fragmentation.
Virtual FAT adds in **long file name support**, that supports long filenames up to 255 characters. It use **multiple directory entries** for a file with long file name. It use a previously invalid file attribute so non-VFAT applications can ignore these additional entries. It also use teh first byte in the filename to indicate sequence.
The $8 + 3$ short version is kept for backward compatibility.

13.2 Extended-2 File System

Disk space is split into Blocks, and blocks are grouped into block groups. Each file/directory is described by a **single** special structure known as I-node(index node). I-node will contain file metadata and data block addresses. **Superblock** descirbes the whole file system, for example total I-nodes number, I-node per group, total disk blocks, disk block sper group. This is duplicated in each block group for redundancy.
Group descriptors describe each block group, for example number of free disk blocks, free I-nodes, and location of bitmaps. It is duplicated in each block group as well.
Block Bitmap keeps track of usage status of blocks of this block group(1 = Occupied and 0 = free)
I-Node Bitmap keeps track kof the usage status of I-nodes of this block group(1 = Occu-



pied and 0 = Free). **I-Node table** is an array of I-nodes, so that each I-node can be access by an unique index, and contains only I-nodes of this block group. Above is the I-Node



Structure, which has size of 128 bytes.
For the data blocks of a directory, it stores a linked list of directory entries for file/subdirectories information within the directory. Each directory entry contains:

- I-node number for that file/subdirectory
- Size of the directory entry, for locating the next directory entry
- Length of the file/subdirectory name
- Type; File or subdirectory

- File/Subdirectory name(up to 255 hracters)

It links to a 0 I-node number to indicate unused entry.
Given a pathname, say `/sub/file`, let `currDir`="/". Root directory has a fixed I-node number 2.
We read the actual I-node and check whether `sub` is a directory or file. If it is the direcotry, retrieve the I-node number and read the actual I-node. Otherwise, locate the directory entry in `CurrDir`, retreive I-node number and read actual I-node.

To delete a file, remove its directory entry rom parent directory. Point the previous entry to next entry/end.
We then update I-node bitmap, by marking corresponding I-node as free. Then we update block bitmap, by mark the corresponding block as free.

For sharing with hard/symbolic link, if it is a hard link, we create a directory entry in B which uses the same I-node number. For symbolic link, creates a new file Y in B where Y contains the pathname of X.