1 Introduction to OS

1.1 Operating System Basic Concepts

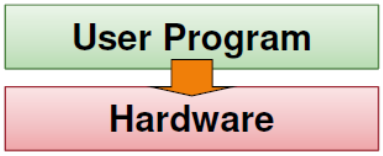
Definition 1.1 (Operating System)**.** An **operating system**(OS) is a program that acts as an intermediary between a **computer user** and the **computer hardware**.

The evolution of OS is as below

No OS → Batch OS → Time-sharing OS → 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:



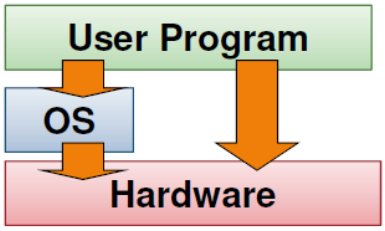
- Minimal overhead

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

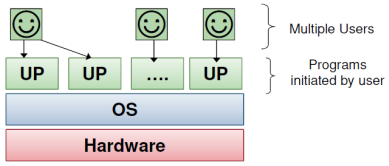


will be idle when I/O. One solution is **multiprogramming**, where multiple jobs are loaded and other jobs are to be ran when I/O needs to be done.

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

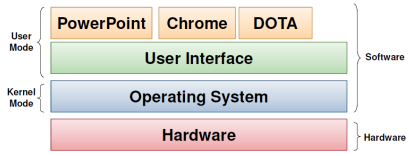
   so that low level details can be hidden and only high level functionality is presented. It provides efficiency and portability.
2. Resource Allocator, which
   - Manages all resources such as CPU, Memory, I/O devices
   - Arbitrate potentially conflicting requests, for efficient and fair resource use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

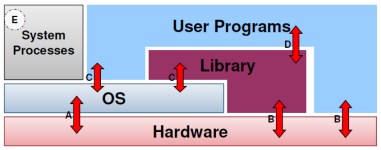
1.3 OS Structure

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

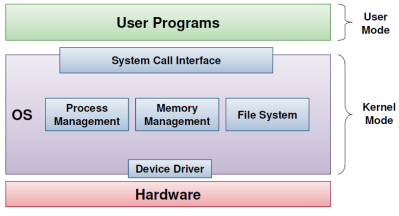
Currently, the common code organisation consists of

- 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

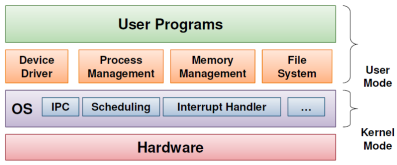
- Highly coupled components
- Usually devolved into very complicated internal structure

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.

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:

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

- 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

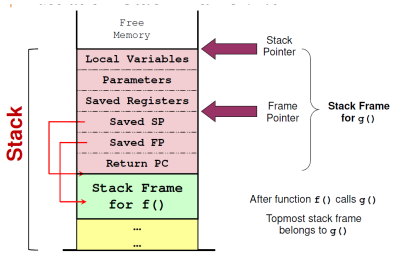
- **Text** section, to store instructions
- **Data** section, to store global variables

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.

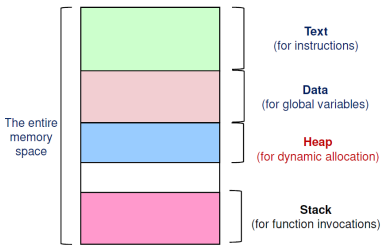
Definition 2.5 (Heap Memory Region)**.** Heap Memory Region is a memory region that supports **dynamically allocated memory**, which is acquisition of memory space during **execution time**.

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

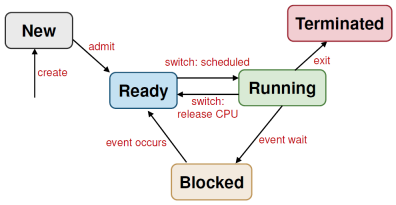
In correspondence to the memory context, the hardware context contains

- 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

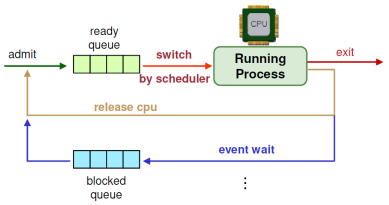
There are also different state transitions,

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/r 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.

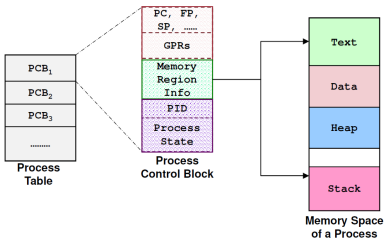
Up to this point, the OS context contains

- 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

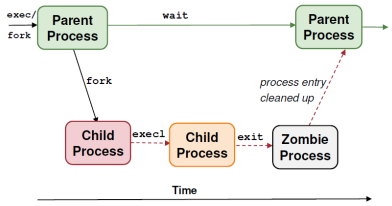
Parent process can wait for child process to terminate, by invoking `wait(*status)`. This call of `wait(*status)` will return the PID of the terminated child process. Also, the address where the `status` pointer points to will be populated with the exit status of the terminated child process. The behaviour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

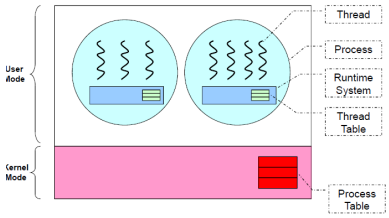
However, thread has the problem of

- 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*



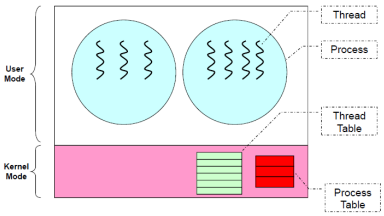
thread model are

- *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

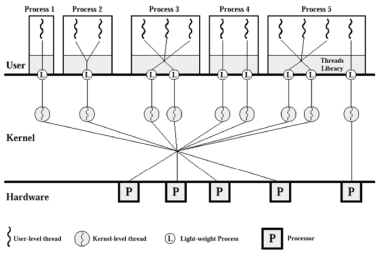
- Kernel can schedule on thread levels, therefore more than 1 thread in the same process can run simultaneously on multiple CPUs.

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))**.**

- `Sen` 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 `Registe`
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 initialize count, in , out to be 0

```
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 Process

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

```
while (TRUE) {

    wait( roomEmpty );

    Modifies data

    signal( roomEmpty );
}
```
Writer Process

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

```
while (TRUE) {

    wait( mutex );
    nReader++;
    if (nReader == 1)
        wait( roomEmpty );
    signal( mutex );

    Reads data

    wait( mutex );
    nReader--;
    if (nReader == 0)
        signal( roomEmpty );
    signal( mutex );

}
```
Reader Process

that favours reader.

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.