CS2106 Introduction To Operating Systems

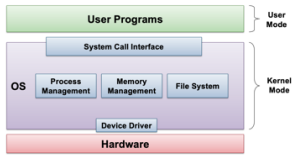
AY2021/22 Semester 2

Introduction

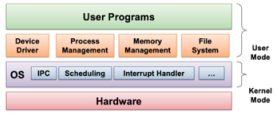
1. Operating Systems

1.1. Introduction to Operating Systems

- OS is a program that acts as an intermediary between a computer user and the computer hardware.
- Monolithic VS Microkernel



Pros:
(1) Well understood
(2) Good performance
Cons:
(1) Highly coupled components
(2) Complicated internal structure

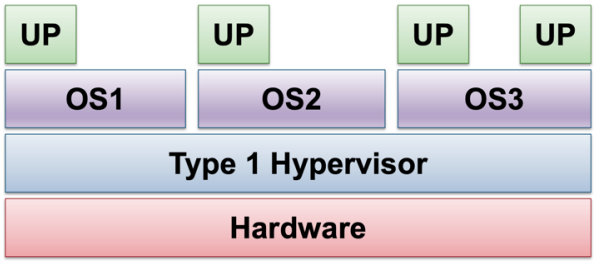


Pros:
(1) More robust and extendable
(2) Better isolation and protection between kernel and high-level services
Cons:
Lower performance

2. Virtual Machines

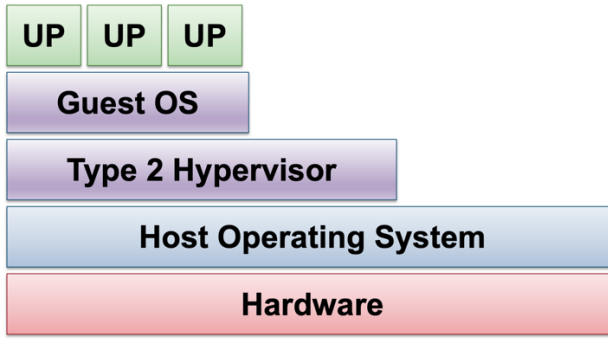
2.1. Type 1 Hypervisor

- Provides individual virtual machines to guest OSes.
- Faster than Type 2 due to less overheads (one fewer layer).



2.2. Type 2 Hypervisor

- Runs in host OS.
- Simpler to build than Type 1.



Process Management

1. Process Abstraction

1.1. Overview

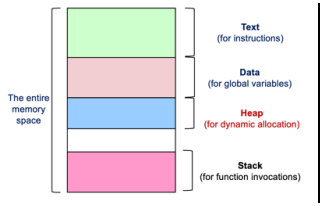
- Process is a dynamic abstraction for executing programs. It includes all information required to describe a running program, including:

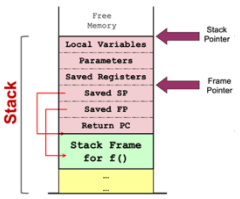
(1) Memory context: Code, data, ...

(2) Hardware context: Registers, PC, stack pointer, frame pointer, ...

(3) OS context: Process properties (PID, state), resource used, ...

1.2. Memory Context





- Stack pointer points to top of stack region.
- **Stack frame setup procedure**

(1) Caller passes params with registers and/or stack.

(2) Saves Return PC on stack.

(3) Transfers control to callee.

(4) Callee saves old SP & FP and registers used by callee.

(5) Allocates space for local variables used by callee.

(6) Adjusts SP to point to the new stack top.

- **Stack frame teardown procedure**

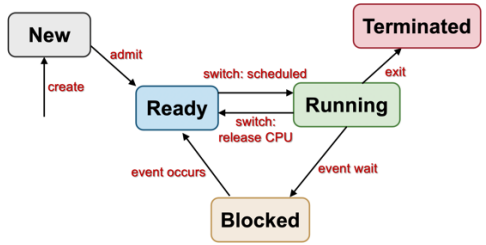
(1) Callee places return result on stack and restores saved SP & FP and saved registers.

(2) Caller utilises return result and continues execution in caller.

- **Frame pointer**: Use of FP is platform-dependent. FP facilitates access of various stack frame items by pointing to fixed locations in stack frames.
- Since the number of general purpose registers (GPR) is limited, a function can spill registers and restore them later.
- malloc() allocates memory in heap section, but the pointer itself is in stack section.

1.3. OS Context

- PID is used for process identification.
- **5-stage model**



(1) NEW: New process created.
(2) READY: Process is waiting to run.
(3) RUNNING: Process is being executed on CPU.
(4) BLOCK: Process is waiting for event.
(5) TERMINATE: Process has finished execution.

1.4. Process Control Block (PCB) and Process Table

- **Kernel** maintains PCB for all processes. A PCB contains execution context for a process, including:

(1) Process state: One of NEW, READY, RUNNING, BLOCK, TERMINATE.

(2) Process number

(3) Program counter: Address of the next instruction.

(4) Registers: Registers that are used by the process, including accumulators, index registers, stack pointers, general purpose registers, etc.

(5) List of open files: Different files that are associated with the process.

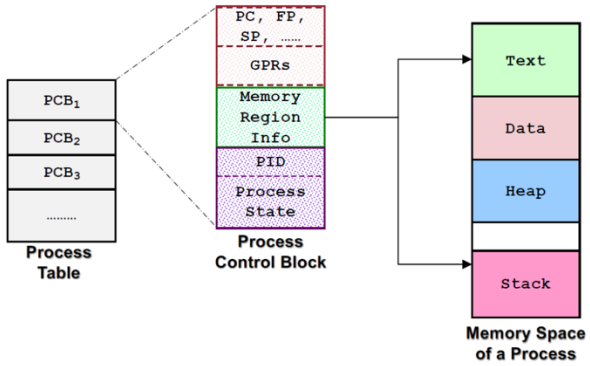
(6) CPU scheduling information: Process priority and other scheduling parameters.

(7) Memory management information: Page tables, segment tables, base registers, limit registers, etc.

(8) I/O status information: List of I/O devices, list of files, etc.

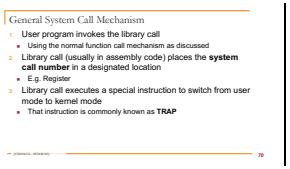
(9) Accounting information: Time limits, account numbers, amount of CPU used, process numbers, etc.

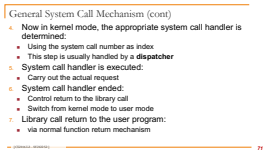
- PCB is located in a memory area that is protected from normal user access (e.g. beginning of kernel stack).



1.5. System Calls

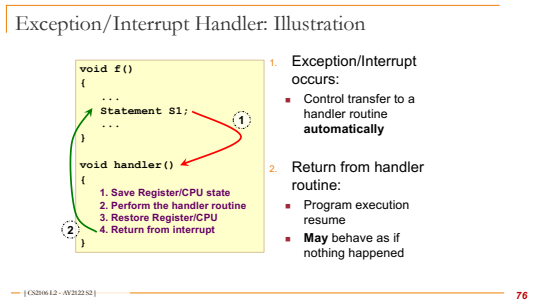
- Syscalls are APIs to the OS. Provides a way of calling facilities/services in the kernel. It is different from normal function call such that it has to change from user mode to kernel mode.





1.6. Exception and Interrupt

- Exception is synchronous, interrupt is asynchronous.
- Exception occurs due to program execution, interrupt occurs independent of program.
- Exceptions have to be handled by exception handler, interrupt will suspend program execution and have to execute an interrupt handler.



1.7. Process Creation in UNIX – fork()

- Returns PID for parent process and 0 for child process.
- Child process is a duplicate of parent process. Parent process and child process have independent memory space.
- Root process is the init process in the kernel at bootup, PID = 1.
- Copy-on-right optimisation

1.8. Process Creation in UNIX – execl(path, args, NULL)

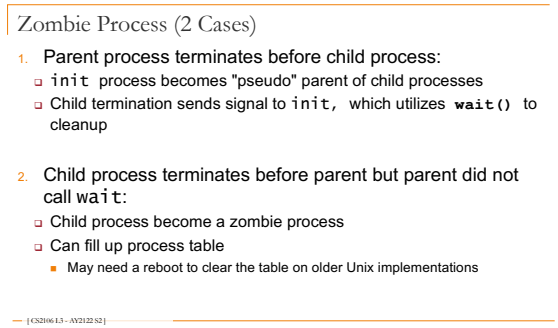
- Replaces current executing process image with a new one.
- Code replacement
- PID and other information are still intact.
- **fork() + execl()**: Spawns off a child process and get parent process ready to accept another request.

1.9. Process Termination in UNIX – exit()

- Returns status to parent process.
- UNIX convention: 0 for normal termination, !0 for problems.
- Releases most system resources (e.g. file descriptor) except PID & status (needed for parent-child synchronisation), process accounting information (CPU time, process table entry may still be needed), etc.
- Becomes zombie.

1.10. Parent-child Synchronisation in UNIX – wait()

- Returns the PID of the terminated child process.
- Blocks parent process until at least one child process terminates.
- If there is no child, continue immediately.
- Cleans up the remainder of child system resources and kills zombie processes.



2. Process Scheduling

2.1. Two Types of Scheduling

- **Non-preemptive scheduling**: Stays scheduled in RUNNING state until it blocks or voluntarily gives up CPU.
- **Preemptive scheduling**: Given a fixed time to run and is suspended at the end of given time.

2.2. Scheduling Algorithms For Batch Processing

- **Evaluating criteria**

(1) Fairness: Ensures fair sharing of CPU time, no starvation.

(2) Balanced utilisation of system resources.

(3) Turnaround time: **Finish_Time – Arrival_Time**

(4) Throughput: Number of finished tasks per unit time.

(5) CPU Utilisation: Percentage of time when CPU is working (compared with I/O).

- **First-come First-servied (FCFS)**: Guaranteed to have no starvation. FCFS minimised average response time if the jobs arrive in the ready queue in order of increasing job lengths.
- **Shortest Job First (SJF)**: Non-preemptive. Minimises average waiting time, but may cause starvation – long jobs never have a chance.
- **Shortest Remaining Time (SRT)**: Preemptive. May cause starvation.

2.3. Scheduling Algorithms For Interactive Processing

- **Evaluating criteria**

(1) Response time: Time between request and response by system. More response means more context switching overhead. Preemptive algorithms are used to ensure good response time.

(2) Predictability: Variation in response time. Less variation means more predictable.

- **Round Robin (RR)**: FIFO with a fixed time slice (quantum). Guarantees response time. If the quantum is

(1) larger, then better CPU utilisation but longer waiting time.

(2) shorter, then shorter response time but bigger overheads due to context switching.

- **Priority Scheduling**: Assigns priority to tasks. May cause starvation. Hard to control the exact amount of CPU time given. Priority inversion when lower priority task pre-empt higher priority task.
- **Multilevel Feedback Queue**: Minimises response time for I/O-bound process and turnaround time for CPU-bound process. New job has the highest priority. If a job fully utilises its time slice, its priority reduces. If a job gives up or blocks before finishing its time slice, its priority retains. Jobs with same priority runs in RR. Favours I/O intensive processes.
- **Lottery Scheduling**: Gives out tickets to processes, and select ticket among eligible tickets. In long run, process holds X% of tickets get scheduled X% of time. Responsive as new processes can participate in next lottery. Prevents starvation.

3. Inter-process Communication

3.1. Shared Memory

- P1 creates a shared memory region and P2 attaches it to its own. In the end, P2 detaches from the region and P1 destroys the region.
- **Advantages**:

(1) Efficient: Only create and attach involves OS.

(2) Ease of use: Only need to create and attach.

- **Disadvantages**:

(1) Limited to a single machine.

(2) Requires synchronisation to avoid data racing.

3.2. Message Passing

- P1 sends a message and P2 receives it. The message must be stored in kernel memory space. Send and receive go through OS by system calls.
- Communication can be direct or indirect (via mailbox).
- **Synchronisation behaviours**

(1) Non-blocking sender: Sender never blocks, even if receiver has not received. If buffer full, sender waits or returns error.

(2) Blocking sender: Sender is blocked until message is received.

(3) Non-blocking receiver: Receiver proceeds empty-handed but does not block.

(4) Blocking receiver: Receiver is blocked until a message has arrived.

- **Advantages**

(1) Applicable beyond a single machine.

(2) Portable.

(3) Easier synchronisation.

- **Disadvantages**

(1) Inefficient.

(2) Hard to use.

3.3. UNIX Pipes

- 1 end for reading, 1 end for writing. Pipes may be unidirectional or bidirectional, depending on UNIX version.
- **Implicit synchronisation**: Writer waits when buffer is full, readers wait when buffer is empty.
- fd[0] is reading end and fd[1] is writing end.

3.4. UNIX Signal

- A form of IPC sent to a process/thread. The recipient must handle the signal by a default set of handlers or user-supplied handlers.
- Async notification regarding event, sent to a process/thread (e.g. kill, stop, continue).

4. Alternative To Process – Thread

4.1. General Idea

- Adds more threads of control to the same process so that multiple parts of the process are executing at the same time. Threads in the same process share memory context (text, data, heap) and OS context (PID, files, etc.). Each thread has unique ID, registers and stack. Only hardware context (registers and SP & FP registers) is switched in thread switch.

4.2. Advantages

- Economic: Much less resources compared with multiple processes.
- Resource sharing: Threads share most of the resources of a process, no need additional mechanism.
- Responsiveness: Much more responsive.
- Scalability: Takes advantages of multiple cores/CPUs.

4.3. Disadvantages

- Synchronisation around shared memory gets even worse (all except stack region).
- System call concurrency
- Process behaviour

4.4. Thread Models

- **User threads**: Implemented as user library. It has many advantages such as

(1) Can have multithreaded programs on any OS.

(2) Thread operations are just library calls.

(3) Generally more configurable and flexible.

However, since OS is not aware of user threads, one thread blocked leads to all threads blocked, and also cannot exploit multiple CPUs.

- **Kernel threads**: Implemented in OS. In this way, kernel can schedule on thread levels, but

(1) Thread operations are system calls (slower and requires more system resources).

(2) Generally less flexible.

- **Hybrid threads**: have both kernel and user threads. OS schedules on kernel threads only, offer great flexibility.

4.5. UNIX Thread

- Pthread creates user thread on Windows and kernel thread on Linux.
- Creation/termination: pthread_exit()/pthread_create()
- Synchronisation: pthread_join() to wait for other threads.

5. Synchronisation

5.1. Race Condition

- Incorrect execution due to the unsynchronised access to a shared modifiable resource.
- **Solution**: Designates code segment for critical section that allows at most one process.
- **Properties of critical section**

(1) Mutual exclusion

(2) Progress: If no process is in the critical section, one waiting process is granted access.

(3) Bounded wait: There exists an upper bound for waiting time.

(4) Independence: Process not executing in critical section does not block other processes.

- **Symptoms of incorrect synchronisation**

(1) Deadlock: All blocked.

(2) Livelock: Keeps changing state and makes no other progress.

(3) Starvation.

(4) Busy waiting: The waiting process repeatedly tests the while-loop condition instead of going into BLOCK state.

5.2. Critical Section Implementations

- **Peterson's Algorithm**

```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
       Turn == 1);
Critical Section
Want[0] = 0;
```

Process **P0**

```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
       Turn == 0);
Critical Section
Want[1] = 0;
```

Process **P1**

- **Semaphore**: $S_{current} = S_{initial}$ + #signal(S) - #wait(S)
- **Mutex**: S = 0 or 1

Wait(S)
- **If S <= 0, blocks** (go to sleep)
- **Decrement S**
- Also known as `P()` or `Down()`

Signal(S)
- **Increments S**
- Wakes up one sleeping process if any
- This operation **never** blocks
- Also known as `V()` or `Up()`

5.3. Synchronisation Problem Examples

- **Producer-consumer**

```
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

- **Reader-writer**

```
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

- **Dining philosophers**

The claim is TRUE. Informal argument below.

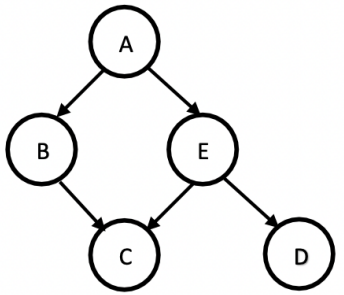
For ease of discussion, let's refer to the right-hander as **R**.

If **R** grabbed the right fork then managed to grab the left fork THEN
R can eat □ not a deadlock.

If **R** grabbed the right fork but the left fork is taken THEN
The left neighbor of R has already gotten both forks □ eating □ eventually release fork.

If **R** cannot grabbed the right fork THEN
The right neighbor of R has taken its left fork. Worst case scenario: all remaining left-hander all hold on to their left fork. However, the left neighbor of R will be able to take its right fork because R is still trying to get its right fork.

- **Constraint graphs**



Line#	Code snippets
1	`// declare any semaphores you need here;`
2	`// be sure to show how to initialize them`
3	`// initial value of all semaphores is 0,`
4	`// for proper code see main function; shorthand is accepted`
…	`sem_t a_done, b_done, e_done;`
	`sem_init(&a_done, 0, 0);`
	`sem_init(&b_done, 0, 0);`
	`sem_init(&e_done, 0, 0);`
A.1.	`static void *thread_A(void *_) {`
A.2.	`printf("A\n");`
A.3.	`sem_post(&a_done);`
…	`sem_post(&a_done); }`
B.1.	`static void * thread_B(void *_) {`
B.2.	`sem_wait(&a_done);`
B.3.	`printf("B\n");`
…	`sem_post(&b_done); }`
C.1.	`static void *thread_C(void *_) {`
C.2.	`sem_wait(&b_done);`
C.3.	`sem_wait(&e_done);`
…	`printf("C\n");`
	`sem_post(&c_done);`
	`}`
D.1.	`static void *thread_D(void *_) {`
D.2.	`sem_wait(&e_done);`
D.3.	`printf("D\n");`
…	`}`
E.1.	`static void * thread_E(void *_) {`
E.2.	`sem_wait(&a_done);`
E.3.	`printf("E\n");`
…	`sem_post(&e_done);`
	`sem_post(&e_done);`
	`}`
M.1.	`int main()`
M.2.	`{`
M.3.	`pthread_t t[N];`
M.4.	`pthread_create(t+0, NULL, thread_E, NULL);`
M.5.	`pthread_create(t+1, NULL, thread_D, NULL);`
M.6.	`pthread_create(t+2, NULL, thread_C, NULL);`
M.7.	`pthread_create(t+3, NULL, thread_B, NULL);`
M.8.	`pthread_create(t+4, NULL, thread_A, NULL);`
M.9.	`pthread_exit(0);`
M.10.	`}`

Memory Management

1. Memory Abstraction

1.1. Memory Usage

- **Types of data in a process**

(1) Transient data (e.g. *function call parameters or local variables*)

(2) Persistent data: Valid during entire program duration unless removed (e.g. *global/constant variables, dynamic allocated memory*).

- Both types of data can **grow/shrink** during execution.

1.2. Memory Management

- **OS memory-related tasks**: allocate + manage memory space for processes; protect memory space from other processes; provide memory-related system calls to processes; manage memory space for internal use.

1.3. Logical Address

- Logical address is how processes view its memory space.
- Generally, logical address ≠ physical address.
- Each process has a self-contained, **independent** logical memory space.

2. Contiguous Memory Allocation

2.1. Memory Partitioning

- **Assumptions**

(1) Each process occupies a contiguous memory region.

(2) Physical memory is large enough to contain ≥1 processes with complete memory space.

- When physical memory is full, OS removes DONE process and swap BLOCK process to secondary storage.
- **Memory partition**: The **contiguous** memory region allocated to a single process.

2.2. Fixed-size Partitions (Fixed-size Partitioning)

- Physical memory is split into fixed number of partitions of same sizes. One process will occupy one partition.
- **Advantage**: Easy to manage and fast to allocate, because each partition is the same and do not need to choose.
- **Disadvantage**: Partition size need to be large enough to contain the largest process, hence smaller processes will waste space, causing **internal fragmentation**.

2.3. Variable-size Partitions (Dynamic Partitioning)

- Partitions are created based on the actual sizes of processes. OS keeps track of the occupied and free regions. Merging and splitting are done when necessary.
- **Advantage**: Flexible and removes internal fragmentation.
- **Disadvantages**

(1) Needs to maintain more information in OS and takes more time to locate appropriate region.

(2) Produces holes when memory is freed, which are not large enough to allocate to another process, causing **external fragmentation**.

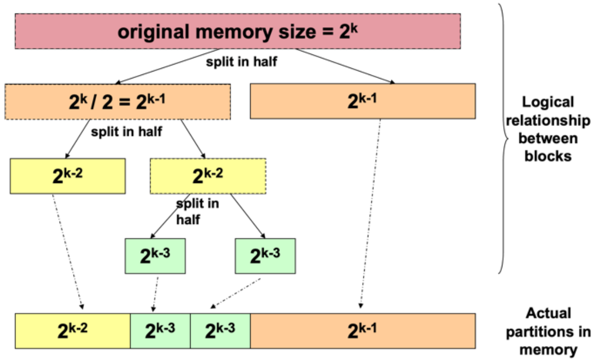
- **Allocation algorithms**

(1) Merging: Merges with adjacent holes if possible.

(2) Compaction: Moves occupied partitions around to create consolidated holes, yet time-consuming.

(3) First-Fit/Best-Fit/Worst-Fit

2.4. Buddy System



- **Data structure**: An array A, each cell A[S] contains a **LinkedList** representing available blocks of size 2^s. A **buddy** is an adjacent block of the same size.
- **Allocation algorithms** ($O(\lg n)$): To allocate block of size N,

(1) Find smallest S such that $2^s \geq$ N.

(2) Access A[S] for a free block. If it exists, then remove it from A[S] and allocate the process; otherwise, find the smallest R > S such that A[R] has a free block.

- **Deallocation algorithms** (worst case $O(n)$): To deallocate a block B,

(1) Check A[S]. If buddy of B, say C, exists, then remove B and C from A[S], merge them to B' ($O(n)$).

(2) Continue to merge B' if possible.

- **Advantages**

(1) Fast allocation.

(2) Size of chosen hole closely matches size of process.

- **Disadvantages**

(1) Can cause internal fragmentation when size of process < partition split.

(2) Can cause external fragmentation when memory space size is not power of 2.

3. Disjoint Memory Allocation

3.1. Page Scheme

- **Assumption**: Physical memory is large enough to contain ≥1 processes with complete memory space.
- Physical memory is split into regions of fixed size (physical frames), decided by hardware. During execution, pages of process load in any available frame.
- Logical memory space remains contiguous while physical memory space may be disjoint.
- **Characteristics**

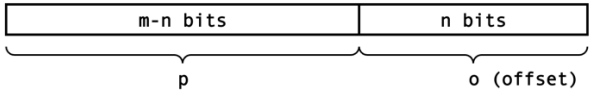
(1) Remove external fragmentation: All available frames can be used.

(2) Still has internal fragmentation: Page size is fixed.

(3) Clear separation between logical and physical address space.

- **Address translation**: Given page size of 2^n, and logical address length of m bits, we have

Physical_Address = Frame_Number x 2^n + o



Frame_Number can be computed from p using mapping mechanism like page table.

- **Page table**: Page table of a process is stored in the RAM in the PCB table of OS. Each process has its unique page table. Hence two memory access for each memory reference (page table + actual memory).
- To check and handle page fault:

(1) [Hardware] Checks page table. If non-memory resident, page fault occurs.

(2) [Hardware] TRAP to OS.

(3) [OS] Global/local replacement.

(4) [OS] Writes out the page to be replaced if needed (if value was modified).

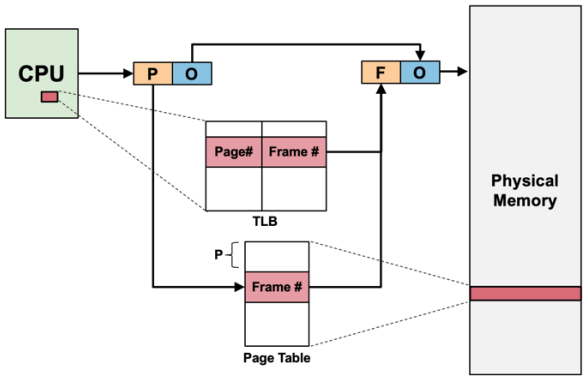
(5) [OS] Locates the page (given page number) in secondary storage.

(6) [OS] Loads the page into a physical frame.

(7) Updates page table entries. Updates TLB.

(8) Returns from TRAP.

- **Translation Look-aside Buffer**: TLB is a specialised on-chip component to support paging – cache of few page table entries. TLB is part of the hardware context of a process. When context switching occurs, TLB is flushed and new process will not get wrong translation, but when the process resumes, there will be many TLB misses. TLB is looked up upon **any memory access** (data/instruction).



- To check and handle TLB fault:

(1) [OS] Access full table of the process (e.g in the PCB of the process), look for page number that did not exist in the TLB.

(2) [OS] Checks whether valid bit is set, otherwise it is segmentation fault.

(3) [OS] Loads the relevant PTE to TLB.

(4) [OS] Returns from TRAP.

- **Protection**

(1) Access-right bits: writable, readable, executable.

(2) Valid bit: Out-of-range access will be caught by OS in this manner.

(3) Page sharing: Copy-on-write.

3.5. Segmentation Scheme

- Separate logical memory space into multiple memory segments. Each segment has a **name** and a **limit**. Memory reference is now **<Segment_Name, Offset>**.
- **Advantages**

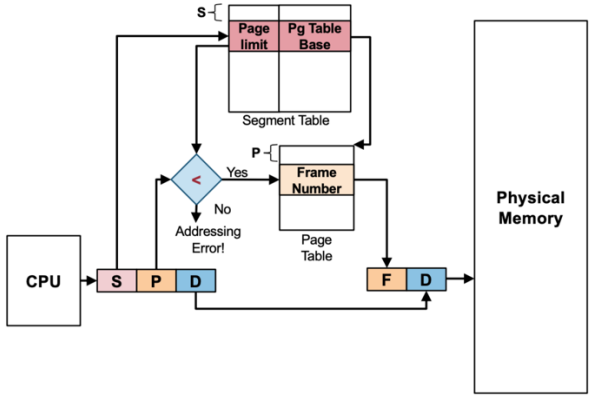
(1) Each segment can grow/shrink independently.

(2) Each segment can be protected/shared independently.

- **Disadvantages**: Segmentation requires variable-size contiguous memory regions, causing **external fragmentation**.
- **Segmentation addressing error: Offset > Limit**

3.6. Segmentation With Paging

- Each segment has a page table and segment can grow and shink.



4. Virtual Memory Management

4.1. Virtual Memory

- Splits logical address space to small chunks – some in physical memory, some in secondary storage.
- **Advantages**

(1) More efficient use of physical memory since unused pages stored on secondary storage.

(2) Allows more processes to reside in memory hence improve CPU utilisation.

- **Mechanism**

(1) Use page table to translate virtual address to physical address.

(2) Distinguish memory resident pages from non-memory resident ones using 1 bit.

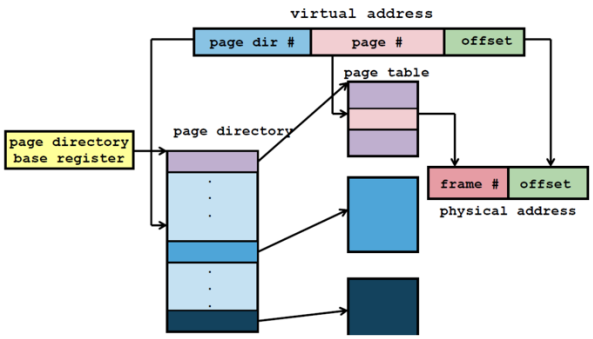
(3) CPU can only access memory resident pages, otherwise **page fault** occurs and **OS** needs to bring pages from non-memory resident pages into physical memory.

- **Locality principles**: To avoid thrashing,

(1) Temporal locality: Memory address used is likely to be used again.

(2) Spatial locality: Memory address closed to a used address is likely to be used.

- **2-level paging**: Used to reduce overheads in unused pages (instead of storing all the pages). Reduces fragmented page table. However, page directory may contain empty entries.



Hierarchical page table is an example of **indirection** design principle – trading off time to save space.

- **Inverted page table**: **<PID, Page_Number>** This answers the question "who is this page shared with?". Also, Solaris OS uses this to translate between PID, Process_Number and Offset to the physical address. Can be implemented by linear scan or hashing. Huge saving by putting everything in one table but slower translation.

4.2. Page Replacement

- **Memory_Access_Time = (1 - P) x T_{mem} + P x T_{Page_Fault}**
- **Optimal page replacement**: Replaces page that will not be used again for longest period of time so that min(Page_Fault). Used as a base of comparison for other algorithms.
- **FIFO page replacement**: Replaces oldest memory page (time first loaded). Simple to implement and no hardware support needed, but does not exploit temporal locality.
- **Least recently used page replacement**: Replaces page that has not been used for the longest period of time. Utilises temporal locality but difficult to implement. Can be implemented by:

(1) A counter: Uses a logical "time" counter which increments for every memory reference. Process Table Entry (PTE) has a "time of use" field which stores time whenever reference occurs. PTE with smallest "time of use" is replaced. However, these requires to search through all pages and the counter may overflow.

(2) A stack: Maintains page numbers, if page is referenced, remove from stack and push on top of stack. Replaces page at the bottom of stack. However, this is not a pure stack and hard to implement in hardware.

- **Second chance page replacement (CLOCK)**: Modifies FIFO to give second chance to pages that are accessed. Each PTE now has a reference bit. If referenced bit = 1, the page is given a second chance but reference bit is set to 0, arrival time reset.
- **Local page replacement**: Victim page selected among pages of the process that causes page fault. In this way, number of frames allocated to a process is constant, leading to stable performances in multiple runs. However, this may hinder the process if number of frames is not enough.
- **Global page replacement**: Victim page selected from the entire physical frames. This allows self-adjustment among processes, but badly behaved processes may affect others and number of frames allocated to a process can be different from run to run.

4.3. Frame Allocation

- **Thrashing**: Heavy I/O to bring non-resident pages into RAM (large memory consumption, low CPU utilisation, slow progress caused by shortage of physical memory).
- **Working set model**: Defines working set window Δ, $w(t, \Delta)$ represents the active pages in interval at time t. Allocates enough frames for pages in $w(t, \Delta)$ to reduce possibility of page fault. With this model, we load all working set pages into frames for that process when a process becomes active (BLOCK to RUNNING). Also can migrate those pages from physical memory to secondary storage when a process becomes inactive (RUNNING to BLOCK).

5. File System Management

5.1. File System

- **General disk structure**: 1-D array of logical blocks.
- A file system generally contains

(1) OS Boot-up information

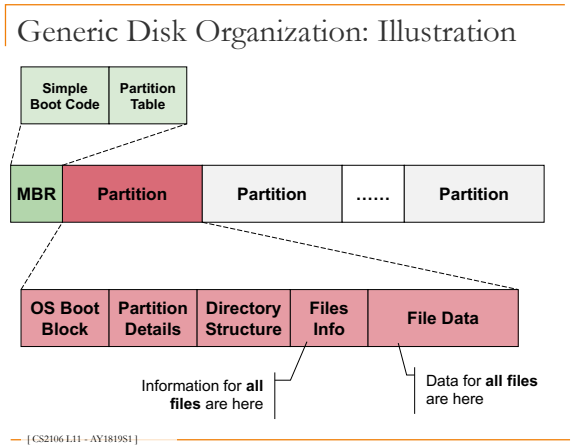
(2) Partition details: Total number of blocks and number and location of free disk blocks.

(3) Directory structure

(4) Files information

(5) Actual file data

- **General disk organisation**



5.2. Implementing Files

- A file is a collection of logical blocks.

- When file size is not multiple of logical block size, the last block may waste space, causing **internal fragmentation**.
- A good file implementation must
    (1) keep track of logical blocks.
    (2) allow efficient access.
    (3) utilise disk space effectively.
- **Contiguous allocation**: Allocates consecutive logical blocks to a file. It has the following advantages:
    (1 Simple to keep track: Each file only needs starting block number and length.
    (2) Fast access: Only needs to seek to first block.

However, it also has some disadvantages:
    (1) External fragmentation: Holes created from file creation/deletion.
    (2) File size needs to be specified in advance.

- **LinkedList Allocation**: Keeps a LinkedList of disk blocks. Each disk block contains the **next** disk block number and actual file data. Each file needs first and last block number. It solves fragmentation problem. However,
    (1) Random access in a file is very slow.
    (2) Part of disk block is used for pointer.
    (3) Less reliable (if one of the pointers is incorrect).
- **File Allocation Table (FAT)**: Move all the block pointers into one single table. FAT is in memory at all time. FAT is simple and efficient used by MS-DOS. The advantage is that the LinkedList traversal is now **in memory**. However, FAT keeps track of all disk blocks in a partition, hence can be huge when disk is large and consume valuable memory space.
- **Indexed Allocation**: Each file has an index block – an array of disk block accesses. The advantages are
    (1) less memory overhead: Only index blocks of opened file needs to be in memory.
    (2) fast direct access

However, the disadvantages are
    (1) limited maximum file size:
Max_Number_Of_Blocks = Number_Of_Index_Block_Entries
    (2) index block overhead

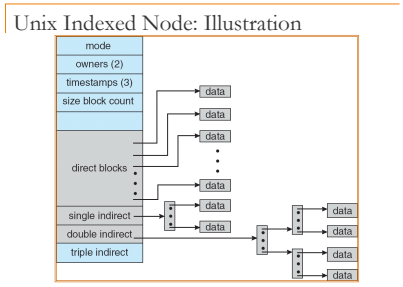


Image taken from "Operating System Concepts" 7th Edition by Silberschatz, Galvin and Gagne
[CS2106 L11 - AY1819S1] 19

5.3. Free Space Management

- Maintains free space information.
    (1) **Allocate**: Remove free disk block from free space list, needed when file is created or enlarged.
    (2) **Free**: Add free disk blocks to free space list, needed when file is deleted or truncated.
- **Bitmap implementation**: Each disk block is represented by 1 bit, where 1 = free and 0 = occupied.

`0 1 0 1 1 1 0 0 1 0 1 1 ......`

- Occupied Blocks = **0, 2, 6, 7, 9, …**
- Free Blocks = **1, 3, 4, 5, 8, 10, 11, …**

This provides a good set of manipulations but needs to keep in memory for efficiency reasons.
- **LinkedList implementation**: Uses a LinkedList of disk blocks. Each block contains the free disk block number and the next block. The advantages are
    (1) easy to locate free blocks.
    (2) only the first pointer is needed in memory, though other blocks can be cached for efficiency.

However, there is high overhead (can be mitigated by storing free block lists in free blocks).

5.4. Implementing Directories

- **Linear list**: Each entry represents a file, which contains:
    (1) File name (minimum) and possibly other metadata.
    (2) File information or pointer to file information.

Locate a file using list requires linear search, which is inefficient for large directories and/or deep tree traversal. A common solution is to use cache to remember latest few searches.
- **Hashtable**: Assures fast lookup but has limited size and depends on good hashing functions.

5.5. Disk I/O Scheduling

- 3 stages of read/write process

3-Stage of Read/Write Process (1/3)

- Time taken to perform a read/write operation:
  = [Seek Time] + [Rotational Latency] + [Transfer Time]

1. Position the disk head over the proper track
   - Time taken is known as [**Seek time**]
   - **Average Seek Time**:
     - Typically in the range of 2 ms to 10 ms
     - (∑ Time for all possible seek) / (Total # of possible seeks)
       ➔ $\frac{1}{3}$ N, where N is the time for maximum seek distance

[CS2106 L11 - AY1819S1] 38

3-Stage of Read/Write Process (2/3)

2. Wait for the desired sector to rotate under the read/write head
   - Time taken is known as [**Rotational latency**]
   - Rotation speed: 4800 to 15000 rotations per minute (RPM)
     ➔ 12.5 ms to 4 ms per rotation respectively
   - **Average Rotational Latency**:
     - Assume desired data is halfway around the track
       ➔ 6.25 ms at 4800 RPM, 2 ms at 15000 RPM

[CS2106 L11 - AY1819S1] 39

3-Stage of Read/Write Process (3/3)

3. Transfer the sector(s)
   - Time taken is known as [**Transfer time**]
   - **Transfer Time** is a function of :
     - Transfer size / Transfer Rate
       - Transfer size: [X KiB / Sector] x [# of Sectors]
       - Transfer rate: 70 to 125 MiB / second
     - E.g. If we read 2 **consecutive sectors** of 512 Bytes
       - Transfer size = 512B x 2 = 1KiB
       - Assuming a transfer rate of 100MiB/second
       - Transfer time = 1KiB / 100MiB per second
         $= 2^{10} / 100 \times 2^{20} = 9.7\mu s$

- (1) + (2) is significantly more than (3)

[CS2106 L11 - AY1819S1] 40

- Algorithms
    (1) First-come First-serve
    (2) Shortest Seek First
    (3) SCAN/Elevator