OS Interaction with Process

Exceptions

- **Synchronous**

1. OS uses exception table to find exception handler (kernel mode)
2. OS executes exeption handler (kernel mode)
3. User process to handle exception/terminate (user mode)

Interrupts

- **Asynchronous**. Cause by external factors
- During interrupts, instructions being executed are atomic and must finish. After that instruction is executed, OS will check for interrupts again

1. Interrupt signal sent to OS (user mode)
2. Save current context (hardware)
3. Disable interrupt (hardware)
4. Hardware reads IVT (stored somewhere in kernel space) to find address of interrupt handler
5. OS invokes interrupt handler (kernel mode)
6. Return from interrupt (user mode)

Syscalls

- **Synchronous**

1. Code invokes syscall (user mode)
2. Syscall sets up syscall number and syscall parameters, and invokes TRAP instruction, raising an exception to switch to kernel mode
3. Dispatcher is invoked with syscall number to find the correct syscall handler using a table similar to IVT, but software version (kernel mode)
4. Syscall handler is invoked (kernel mode)
5. Return from syscall (user mode)

Process Abstraction in Unix

fork()

- Creates an exact duplicate of the parent (except PID and PPID) with completely different addressing space. Global variables, pointers and address pointed to by pointers are copied as well
- Copy on write: read-only data is not copied. Only data that is to be written either by parent or child is copied by child

exec()

- Replaces current process image with new process image. If the process is a child of a parent, it still remains as a child.

wait()

- When a child exits before parent → child can become **zombie** process if parent has not called wait()
- Parent calls **wait()** to clean up child process (PID, status, CPU time)
- wait() only waits for direct children (not grandchildren)
- When parent exits before child → child becomes **orphan** process. "init" process becomes pseudo-parent. Child sends signal to "init" which cleans it up

Inter-process Communication

Shared memory (implicit)

1. Process A creates shared memory region M (OS)
2. Process B attaches M to its own memory space (OS)
3. Processes A and B can read and write into M
4. Detach M from memory space after use (one of the processes)
5. Destroy M if M is not attached to any process (one of the processes)

- OS only involved in **creating** and **attaching** M
- Efficient, easy to use and fast
- Limited to a single machine and requires synchronisation

Message Passing (explicit)

- **Direct**: Sender and receiver explicitly name the other party (via PID)
  - Used when it matters which process receives the message
- **Indirect**: Messages are sent to/received from mailbox/port via syscalls
  - Used when it does not matter which processes received which messages in the mailbox/port
- Assumed **blocking** receiver: receiver needs to wait until it receives a message
- **Asynchronous**: sender is non-blocking
  - System Buffer buffers incoming messages at the receiver
  - Sender is blocked or returns with error when buffer is full
- **Synchronous**: sender is blocking
  - Sender is blocked until receiver calls receive()
  - No need for buffer
  - Message is kept by the sender until receiver copies it directly from sender
- Applicable beyond a single machine and easier to synchronise
- Ineffficient (OS is heavily involved) and harder to use

Unix Pipes

- Asynchronous message passing
- Behaves in FIFO order
- Writers wait when buffer is full and readers wait when buffer is empty

Unix Signals

- Asynchronous notification regarding an event
- Messages sent to a running process to trigger specific behaviours
- Recipient of the signal must handle the signal by default set of signal handlers or custom defined handlers (some signal handlers cannot be overridden)

Synchronisation

Correct CS implementation

- **Mutual Exclusion**: At most one process in CS
- **Progress**: If no process in CS, one waiting process should enter CS
- **Bounded wait**: After a process requests to enter CS, there exists an upper bound on the number of other processes that can enter CS before it
- **Independence**: Process not executing in CS should never block other wanting to enter

Spinlock/Busy-waiting

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**

- Want must be before Turn
  - P0: Turn = 1 → P1: Turn = 0 → P1: Want[1] = 1 → **P1: Enter CS** → P0: Want[0] = 1 → **P0: Enter CS**
- Wasteful: processes that are busy waiting still utilise CPU

TestAndSet

- Atomic function that takes a memory address M, returns current content in M, then set content in M to 1

```
void EnterCS( int* Lock )
{
   while( TestAndSet( Lock ) == 1);
}
```

```
void ExitCS( int* Lock )
{
   *Lock = 0;
}
```

- There's still busy waiting

Semaphores

- Guaranteed no deadlocks if processes lock mutexes in the same order; the order of unlocks does not affect the correctness
- If two processes use 1 mutex, deadlocks are impossible

(Unreusable) Barrier Semaphore

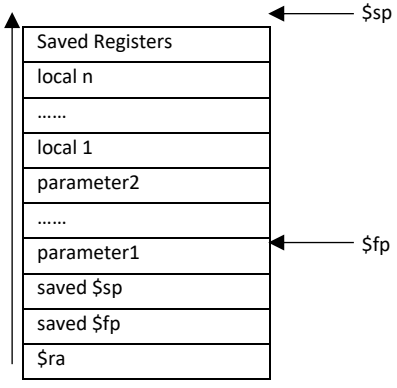
```
int arrived = 0;
Semaphore mutex = 1;
Semaphore waitQ = 0;
Barrier(N) {
   wait(mutex);
   arrived++;
   signal(mutex);
   if (arrived == N) {
      signal(waitQ);
   }
   wait(waitQ);
   signal(waitQ); // final waitQ >= 1
}
```

Operating Systems

- **Monolithic** kernel: One big program; good performance
- **Microkenel**: Small, containing only basic and essential facilities. Excludes device drivers, file systems etc. Provides better isolation and protection and more robust
- **Type 1 Hypervisors**: Runs on top of hardwardware and must implement entire interface to hardware. Harder to implement but faster than Type 2.
- **Type 2 Hypervisors**: Runs on top of host OS and does not communicate with hardware directly. Easier to implement but slower than Type 1.

Process Abstraction

Stack Frames



- **Memory** context: Text, Data, Stack, Heap
- **Hardware** context: GPRs, PC, $sp, $fp
- **Set-up**: Layout and size of stack frame is determined at compile time → address of local variables and parameters will always have the same (relative) address. Reserve space for stack by incrementing $sp
- main's stackframe does not contain saved registers → invoked by OS, registers are saved in PCB
- **Frame Pointer**: Not necessary, but is there for convenience of compiler. Allow usage of constant offset from $fp to address variables on the stack
- **Stack Pointer**: Can change during run-time, due to VLAs
- **Teardown**: Restore saved registers, $sp and $fp. Memory previously allocated for the stack is not deallocated (tearing down is just restoring register values)

Dynamically Allocted Memory

- Allocated in run time. Resides in **heap** memory region
- Cannot place in Data region: size unknown in compile time
- Cannot place in Stack region: need to persist after function returns

Process Model

- **OS** context: Process ID, Process state
- **PCB**: Stores entire execution context: Memory, Hardware and OS

Process Scheduling

- Criteria: (i) Fairness, (ii) Utilisation (minimise idling)
- **Turnaroud time**: Waiting time + CPU Time + I/O time; arrival to finish
- **Throughput**: # tasks finished per unit time
- **Makespan**: Total time to process all tasks
- **CPU utilisation**: Percentage of time when CPU is working on a task

Batch Processing

First Come First Serve (FCFS)

Tasks acquire CPU in FIFO manner. Non-preemptive; runs until done or blocked

Advantages
- No starvation

Disadvantages
- Long waiting time

Convoy effect: CPU-heavy task *T* arrives first, blocking I/O-heavy tasks from CPU →I/O idles. When *T* enters I/O phase, I/O-heavy tasks quickly finished their CPU-phase and joins I/O queue, waiting for *T* → CPU idles

Shortest Job First (SJF)

Choose process with smallest total CPU time. Non-preemptive.

CPU time prediction: $Predicted_{n+1} = \alpha Actual_n + (1- \alpha)Predicted_n$

Advantages
- Guarantees shortest waiting time

Disadvantages
- Can cause starvation – new tasks with shorter CPU time can keep arriving
- Need to predict CPU time for each process, which may not be accurate

Shortest Remaining Time (SRT)

Choose process with shortest remaining CPU time. Preemptive variant of SJF i.e. arrival of tasks with shorter remaining CPU time will preempt current task immediately. No time quantum

Advantages
- Little overhead; decision making only when a task arrives/completes
- Guarantees shortest possible average waiting time

Disadvantages
- Can cause starvation – new tasks with shorter CPU time can keep arriving
- Need to predict CPU time for each process, which may not be accurate

Interactive Systems

- Short TQ: Shorter waiting time, larger context switch overhead, lower CPU utilisation
- Long TQ: Longer waiting time, smaller context switch overead, higher CPU utilisation
- Short ITI: More timer interrupts, less time spent on actual user processes, longer turnaround time

Round Robin

Preemptive variant of FCFS. Task runs until TQ expires, is blocked, or is completed

Advantages
- No starvation
- Reponse time guaranteed to be at most (n-1)TQ

Disadvantages
- If CPU time for tasks are the same and much greater than TQ → longer average TaT than FCFS
- If many tasks and their CPU time is much greater than TQ → context swtich overhead → reduced throughput

Priority Scheduling

Choose process with highest priority. Non-preemptive and preemptive.

In non-preemptive variant, newly arrived tasks with higher priority need to wait until current task gives up CPU. In preemptive variant, arrival of higher priority task preempts task in CPU **even before its TQ expires**

Advantages
- High priority tasks get CPU quicker
- Responsive

Disadvantages
- Low priority tasks can be starved by the continuous arrival of higher priority tasks

Multi-level Feedback Queue (MLFQ)

Newly arrived tasks have high priority. Higher priority tasks can preempt lower priority task **even before its TQ expires**. If TQ is fully utilised, a task is demoted to the next lower priority queue, otherwise, its priority is retained. For tasks with same priority, they are scheduled in RR fashion.

Advantages
- Responsive

Disadvantages
- Can be abused by blocking a task just before its TQ expires to retain its high priority → starve low priority processes
- Task with length CPU phase followed by I/O phase → degrades responsiveness since it sinks to the lowest priority during CPU phase

Lottery Scheduling

Allocate each task a number of "tickets". Pick a ticket randomly and the corresponding task holding that ticket gets scheduled

Advantages
- Responsive, since newly arrived tasks can participate in next lottery
- Good level of control; number of tickets per task can be controlled

Producer-Consumer Semaphores

- Producers: Wait when buffer is full
- Consumers: Wait when buffer is empty

Busy waiting

```
while (TRUE) {
    Produce Item;
    while(!canProduce){}
    wait( mutex );
    if (count < K) {
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        canConsume = TRUE;
    } else
        canProduce = FALSE;
    signal( mutex );
}
```
Producer Process

```
while (TRUE) {
    while (!canConsume){}
    wait( mutex );
    if (count > 0) {
        item = buffer[out];
        out = (out+1) % K;
        count--;
        canProduce = TRUE;
    } else
        canConsume = FALSE;
    signal( mutex );
    Consume Item;
}
```
Consumer Process

- `canProduce` / `canConsume`: Ensures no starvation (at most n producers
  before consumers can consume, vice versa)
- Busy waiting → CPU wastage

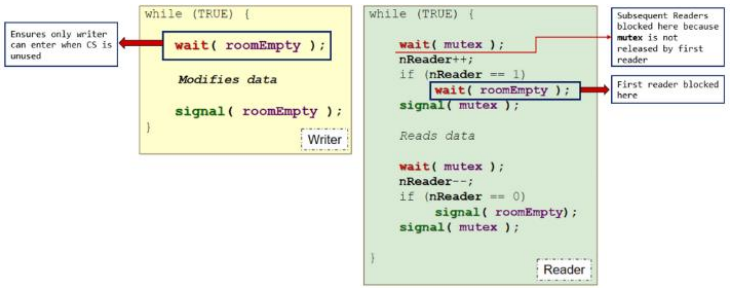
```
while (TRUE) {
    Produce Item;

    wait( notFull );
    wait( mutex1 );
    buffer[in] = item;
    in = (in+1) % K;
    signal( mutex1 );
    signal( notEmpty );
}
```
Producer Process

```
while (TRUE) {

    wait( notEmpty );
    wait( mutex2 );
    item = buffer[out];
    out = (out+1) % K;
    signal( mutex2 );
    signal( notFull );

    Consume Item;
}
```
Consumer Process

- Initialise `notFull = buffer_size, notEmpty = 0`
- Used in asynchronous message passing

Readers-Writers Semaphores

- Exclusive rights for writers. Simultaneous readers



- Problem: Constant stream of readers → nReader will never be 0 → `roomEmpty` never released → starve writers (cut writers' queue)
- Add `revDoor` semaphore

```
while (TRUE) {
    wait(revDoor);
    wait( roomEmpty );

    Modifies data

    signal( roomEmpty );
    signal(revDoor);
}
```
Writer

```
while (TRUE) {
    wait(revDoor);
    signal(revDoor);
    wait( mutex );
    nReader++;
    if (nReader == 1)
        wait( roomEmpty );
    signal( mutex );

    Reads data

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

}
```
Reader

- As soon as there is a writer waiting on `roomEmpty`, no readers can wait on `roomEmpty`

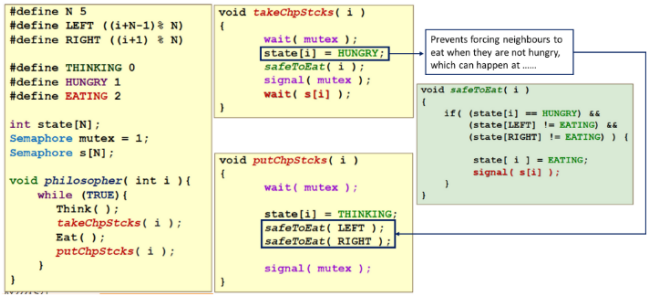
Dining Philosophers

Limited Eater

- Initialise `seats = N - 1`
- Guarantees at least 1 philosopher can have both chopsticks

```
Semaphore seats = S(4);
//initialization

void philosopher( int i ){

    while (TRUE){
        Think( );
        wait( seats );
        takeChpStck( LEFT );
        takeChpStck( RIGHT );
        Eat( );
        putChpStck( LEFT );
        putChpStck( RIGHT );
        signal( seats );
    }
}
```

Tanenbaum Solution



- Each philosopher waits for its own private semaphore

Threads

- Shared among threads: Text, Data, Heap, Resources (I/O, Files)
- Not shared among threads: Stack, CPU Registers
- Threads from the same process can run on different CPUs

Threads	Processes
Switch: involves hardware context (changing FP, SP, register values)	Context switch: involves hardware, OS, memory context
Multiple threads in same process eequires less resources to manage vs multiple processes	Requires more resources to manage

- Syscall concurrency: parallel syscalls possible (e.g. multiple threads can open the same file. No thread will lock a file)

Operation	Behaviour (Linux)
`fork()`	Child process will only inherit the thread on which fork() was called
`exit()`	Entire process is killed (all threads killed)
`exec()`	Image of entire process is replaced (all other threads killed)

- Single-core CPU with simultaneous multithreading (SMT), or hyperthreading, can execute multiple threads at the same time

Thread Models

User Threads	Kernel Threads
Implemented as user library	Implemented and handled by the OS
Kernel unaware of these threads	Kernel aware of these threads
Thread level scheduling by OS not possible	Thread level scheduling by OS possible
Less syscalls, customisable	More syscalls, not customisable
Cannot exploit multi-core CPUs. When one thread is blocked, the OS sees the entire process as blocked → remaining threads cannot run on the other CPUs	Can exploit multiple CPUs. When one thread is blocked, the remaining threads can run on other CPUs

Hybrid Thread Model

- User thread binds to a kernel thread

Memory Management

Memory Abstraction

- **Logical address**: Program's view of the memory
- **Physical address**: Actual address in hardware

Address Relocation

- Recalculate all memory references (increment by *b*) when program loads
- **Base register**: Stores base *b*, physical memory address at start of program
- **Limit register**: Stores the range of memory address for the process
- Addressing:
  1. Physical addr = *b* + MemRef
  2. Check Physical addr < Limit
- Operations done by hardware → Fast

Contiguous Memory Management

- Memory is divided into partitions
- Entire process fits into one partition
- **Internal Fragmentation**: Memory wastage inside partition
- **External Fragmentation**: Memory wastage outside of partitions

Fixed size partitioning

- All partitions are of fixed and equal size
- One partition, one process. Processes cannot use space caused by internal fragmentation

Pros	Cons
Easier to manage	**Internal Fragmentation**
Fast; all free partitions are equal. Just pick any free partition	Need to ensure that partition size needs to be large enough to fit largest process

Dynamic partitioning

- Partition size = process size

Pros	Cons
No internal fragmentation	**External Fragmentation** – when one process is freed, it leaves a whole between two processes
	Due to EF, reasonably sized processes may not fit into any of the holes
	Takes time to find a partition
	More OS memory overhead

- **First-Fit**: Take first partition that is large enough. Fast
- **Best-Fit**: Take smallest partition that is large enough
- **Worst-Fit**: Take largest partition
- **Optimisations**:
  - Merge adjacent holes
  - Compaction: occasionally move occupied partitions around to create larger partitions

Multiple Free Lists (Buddy System)

- Array of linked-lists. Each linked-list is a list of free fixed size partitions, each typically of size 2^n
- Allocation Steps (Process size = N):
  1. Find largest m s.t. $2^m \geq N$
  2. Access A[m]: Partition in A[m]
     - Available: Allocate first partition in list
     - Unavailable: Find smallest partition in A[m+1:] and recursively split until you get a partition of size A[m]
- Deallocation Steps (Process size = N):
  1. Add partition back to A[m]
  2. If buddy is free: Recursively,
     - merge and place merged partition in A[$m + 1$]
     - $m \leftarrow m + 1$
- Two partitions are buddies if they are equal in size (2^m) and only the m-th bit (zero-indexed) differs

Disjoint Memory Management

- Logical memory is broken down into pages/segments which can occupy various parts of physical memory

Paging

- Logical memory broken down into fixed size pages
- Physical memory broken down into same sized frames. Size is determined by **hardware**
- Logical pages fit into any physical frames
- **Page Table (PT)** maps logical pages to physical frames. Stored in kernel space
- Pointer to PT is part of memory context of a process
- **Page Table Entry (PTE)**: Frame Number, Access-rights bit, Valid-Invalid bit
- Mapping (Page size = 2^n, m-bit address):
  1. Page P = most signficant $m - n$ bits of logical address
  2. Frame $F = P^{th}$ PTE
  3. Physical Address = $F(2^n)$ + Offset

Pros	Cons
No external fragmentation	Every memory reference (including instructions) requires 2 memory accesses → slow
Negligible internal fragmentation – at most 1 internally fragmented frame per process	

Paging with Caching

- **Translation Look-Aside Buffer (TLB)** caches the most recently accessed PTEs
- TLB is a hardware component which the OS is aware of
- Reduces number of PT access → reduces performance overhead of logical-to-physical addr translation and reduces
- TLB not part of process' context
- **TLB Entry:** Page Number, Frame Number, Access-rights bit, Valid-Invalid bit
- Mapping (Page size = 2^n, m-bit address):
  1. Page P = most signficant $m - n$ bits of logical address
  2. TLB[p]?
     - TLB Hit: Frame F = TLB[p].Frame. Done
     - TLB Miss: Frame $F = P^{th}$ PTE
  3. Physical Address = $F(2^n)$ + Offset
- Average Memory Access Time (AMAT):
  $P(\text{TLB Hit}) \times (T_{TLB} + T_{Mem}) + P(\text{TLB Miss}) \times (T_{TLB} + 2 \times T_{Mem})$
- TLB contents **flushed** during context switch → prevent incorrect addr translation for next process

Page Sharing

- Use same physical frame number in PTEs
- **Copy-on-Write**: Child process has same PT contents as parent until it writes into a page → change Frame Number for written page
- Shared code page (e.g. C stdlib, syscalls)

Segmentation

- Logical memory broken down into variable sized segments
- Each segment occupies a contiguous physical memory space
- **Segment Table (ST)** maps segments to physical memory. Stored in kernel space
- **Segment Table Entry**: Limit, Base
  - Limit: Max size of segment
  - Base: Starting physical addr of segment
- Permissions are scoped per segments (vs page in paging)
- Gaps between segments are provisioned as segments can grow/shrink
- Mapping:
  1. Segment Entry $S = S^{th}$ entry on ST
  2. Check Offset < S.Limit
  3. Physical Address = S.Base + Offset
- Segment Table can be cached in CPU registers → no need lookup ST

Pros	Cons
No internal fragmentation	**External Fragmentation**
Efficient book keeping (information is scoped per segment) → less duplicated informtion	
Matches programmer's view of memory	

Segmentation with Paging

- Logical memory broken down into variable sized segments
- Each segment is further broken down into fixed size pages
- Each segment has its own page table
- Segment occupies non-contiguous physical memory space
- **Segment Table Entry**: Page Limit, Page Table Base, Permission bits
  - Page Limit: Max number of pages
  - Page Table Base: Physical address of segment's page table
- Mapping (Page size = 2^n):
  1. Segment Entry $S = S^{th}$ entry on ST
  2. Check Page P < S.PageLimit
  3. Frame $F = P^{th}$ PTE in PT pointed by S.PageTableBase
  4. Physical Address = $F(2^n)$ + Offset

Pros	Cons
No external fragmentation	
Negligible internal fragmentation – at most 1 internally fragmented frame per segment	

Virtual Memory Management

- Logical memory > physical memory
- Store some pages in secondary storage
- **Resident Bit** in PT: Indicates if a page is in physical memory

Demand Paging

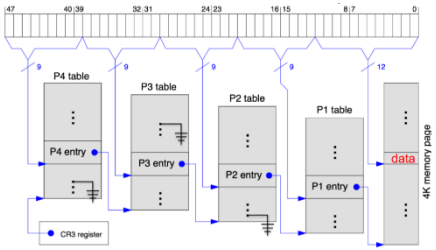
- All pages initially on disk. Bring pages into memory only when needed
- Memory Access (Page size = 2^n, m-bit address):
  1. Page P = most significant $m - n$ bits of logical address

2. TLB[p]?
   - TLB Hit: Frame F = TLB[p].Frame. Go to step 4
   - TLB Miss: Step 3
3. PT[p].Resident?
   - Resident: Frame F = PT[p].Frame
   - Non-Resident: **Page Fault**. Repeat from step 1
4. Physical Address = $F(2^n)$ + Offset

- **Page Fault**: Accessing a page that is not in physical memory
- Page Fault procedure:
  1. Process is BLOCKED. TRAP and enters kernel mode
  2. DMA Controller locates page in disk
  3. DMA Controller loads page onto physical memory
     - RAM is full: Page Replacement Algorithm to evict a page
  4. Update PT for loaded page (Resident bit, frame #)
     - RAM was full: Update PT for evicted page
  5. Exit kernel mode

Pros	Cons
Fast startup time	Process appears sluggish at the start
Small memory footprint	Page faults have cascading effect on other processes (**thrashing**)

Hierarchical Paging
- Reason: Large VM space → large number of pages → large PT → entire PT resides in memory, even if a small subset of pages is used by a process → huge wastage
- Calculate n # levels (v-bit VM addr, r-bit RAM, 2^p page size, 2^e PTE size):
  1. Total # pages in VM = 2^{v-p}
  2. Total # frames in RAM = 2^{r-p}
  3. # PTEs in physical frame = 2^{r-p-e}
  4. $(2^{r-p-e})^n = 2^{v-p}$
  5. $n = \lceil v\text{-}p \,/\, r\text{-}p\text{-}e \rceil$
- Memory Access (m-bit logical add, n-level paging, 2^e entries per PT/PD, Page size = 2^p):
  1. Page P = most significant $m - en$ bits of logical address
  2. TLB[p]?
     - TLB Hit: Frame F = TLB[p].Frame. Go to step 4
     - TLB Miss: Step 3
  3. Repeat n times (**Page-table Walk**):
     3.1. i = [m: m - e] (i.e. most siginificant e bits starting from m)
     3.2. PT_{n-1}/Frame F = $PT_n[i]$.Frame
     3.3. $n \leftarrow n-1$, $m \leftarrow m-e$
  4. Physical Address = $F(2^p)$ + Offset



- Page walks are done in hardware
- **MMU cache** caches most recently accessed PD entries
- Root PT is part of memory context of a process

Pros	Cons
Saves memory space – PDs can have empty entries; only store in memory the PTs containing pages used by the process	Requires multiple serialised memory accesses (page-table walk)

Inverted Page Table
- <Frame #> → <PID, page #>
- # entries = # frames in physical memory → hence smaller than PTs
- Mapping (Frame size = 2^n):
  1. Lookup hashtable[PID, page #] to get F (<PID, page#> is F^{th} entry in inverted PT)
  2. Physical Address = $F(2^n)$ + Offset

Page Replacement Algorithms

- **Dirty Bit** in PT: indicates whether a page has been written into since the last time it was loaded into memory
  - Clean page: just evict, no need to write back to disk
  - Dirty page: need to write back to disk

Optimal Page Replacement (OPT)
- Not realisable, but good as base of comparison
- Replace the page that will not be needed again for the longest period of time
- Guarantees minimum # of page faults

FIFO
- Replace the oldest page
- OS needs to maintain a queue of resident page numbers
- Simple to implement (no hardware support needed)
- Suffers from **Belady's Anomaly** (does not exploit temporal locality)
  - # page faults can increase with more physical frames

Least Recently Used (LRU)
- Replace the page that has not been used in the longest time
- Explots temporal locality
- **Counter** Approach:
  - PT stores "time-of-use" field which increments when a page is referenced
  - Replace the page with smallest "time-of-use"
  - Linear search on PT + "time-of-use" is monotonically increasing → can cause overflow
- **Stack** Approach:
  - Maintain a stack of page numbers
  - Page X is referenced → remove X from stack and push it on top of stack
  - Replace the page at the bottom of the stack
  - Not a pure stack + Hard to implement in hardware

Second-chance Page Replacement (CLOCK)
- FIFO, but skip the page if it has been accessed since the previous round of replacement
- **Reference Bit** in PT: indicates whether a page has been accessed since the previous round of page replacement
- Reference Bit is toggled by hardware
- Algorithm:
  1. Inspect oldest page X (FIFO)
  2. If reference bit == 0: Replace page X. Done
  3. If reference bit == 1: Skip page X and set PT[X].Reference = 0
  4. Repeat from step 1 until a page is chosen
- If all pages have reference bit == 1, it degenrates into FIFO