Layer 1: Classic OS Problems Analysis – Comprehensive Report

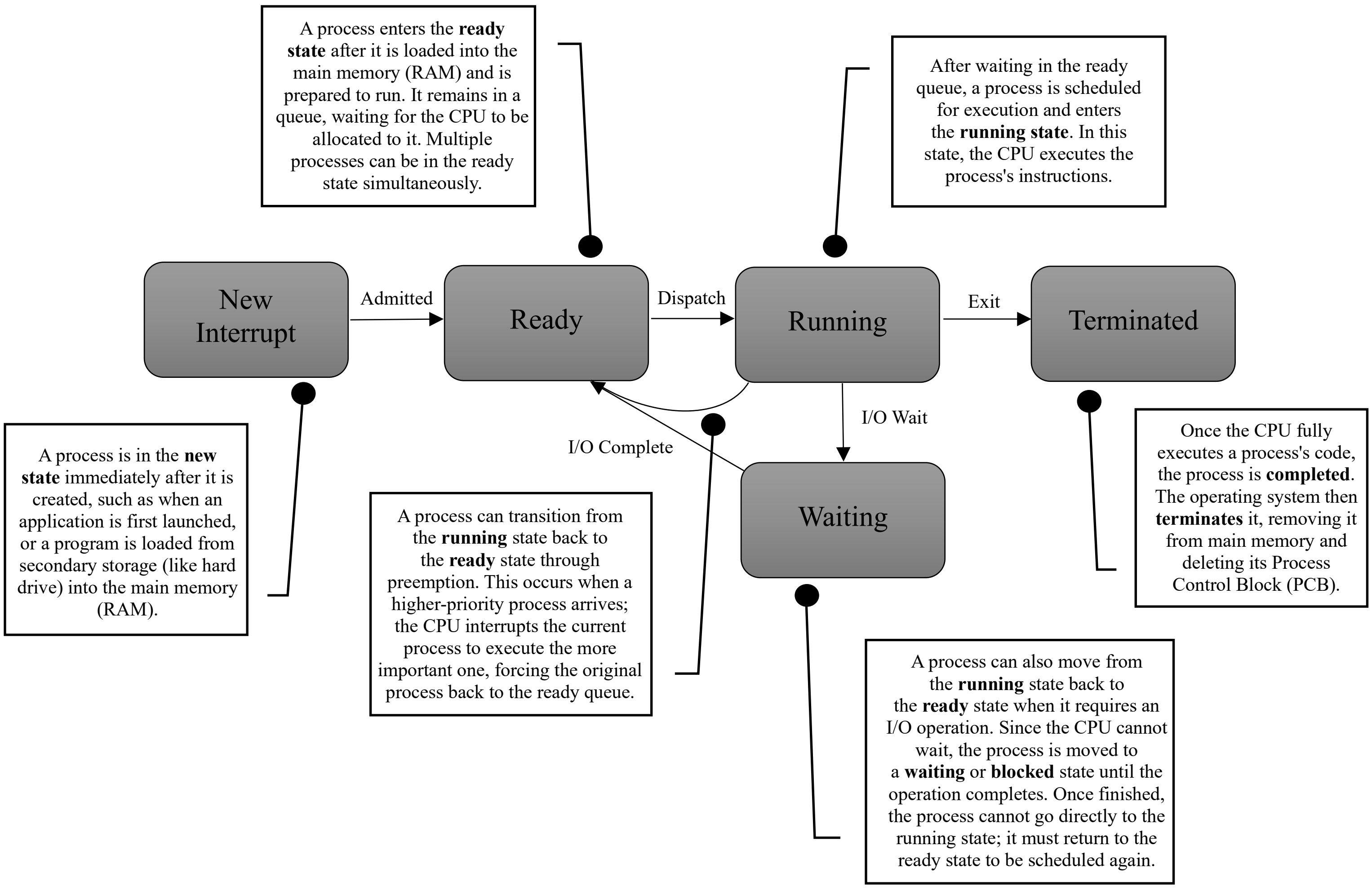
Name: Donna Ritz C. Deguit

PART A: Process Management Fundamentals

- Comparison Table: Process vs. Threads

PROCESS	THREADS
✓ An instance of a program in execution.	✓ Unit of execution within a process.
✓ Processes have separate memory spaces.	✓ Threads within a process share a memory address space.
✓ One process cannot corrupt the memory space of another process.	✓ One misbehaving thread could bring down the entire process.
✓ Handled by context switching when run on a CPU (slow and expensive)	✓ Handled by context switching when run on a CPU (faster)
✓ Communication between processes requires Inter-Process Communication (IPC), which is complex.	✓ Threads can communicate directly via shared memory, which is simple.

- Process State Diagram



A process enters the **ready state** after it is loaded into the main memory (RAM) and is prepared to run. It remains in a queue, waiting for the CPU to be allocated to it. Multiple processes can be in the ready state simultaneously.
After waiting in the ready queue, a process is scheduled for execution and enters the **running state**. In this state, the CPU executes the process's instructions.
New Interrupt
Admitted
Ready
Dispatch
Running
Exit
Terminated
I/O Wait
I/O Complete
Waiting
A process is in the **new state** immediately after it is created, such as when an application is first launched, or a program is loaded from secondary storage (like hard drive) into the main memory (RAM).
A process can transition from the **running** state back to the **ready** state through preemption. This occurs when a higher-priority process arrives; the CPU interrupts the current process to execute the more important one, forcing the original process back to the ready queue.
Once the CPU fully executes a process's code, the process is **completed**. The operating system then **terminates** it, removing it from main memory and deleting its Process Control Block (PCB).
A process can also move from the **running** state back to the **ready** state when it requires an I/O operation. Since the CPU cannot wait, the process is moved to a **waiting** or **blocked** state until the operation completes. Once finished, the process cannot go directly to the running state; it must return to the ready state to be scheduled again.

Layer 1: Classic OS Problems Analysis – Comprehensive Report

- Inter-Process Communication (IPC) Methods

1. **Message Passing -** processes exchange discrete messages through a dedicated channel (like a mailbox) provided by the OS, eliminating the need for shared memory. It operates through two main models:
   - **Synchronous (Blocking)** requires both the sender and receiver to wait for each other, like a live phone call. **Asynchronous (Blocking)** allows the sender and receiver to operate independently, like sending an email.

Email is one of the real-world examples of asynchronous message passing.

2. **Shared Memory -** processes access a common block of RAM, avoiding the need for system calls during data access. However, this speed comes with a trade-off: processes must use synchronization mechanisms like semaphores to prevent race conditions and data corruption when reading or writing to the shared area.

A shared whiteboard in an office or a collaborative document in Google Docs is one of the examples of shared memory.

PART B: Memory Management Basics

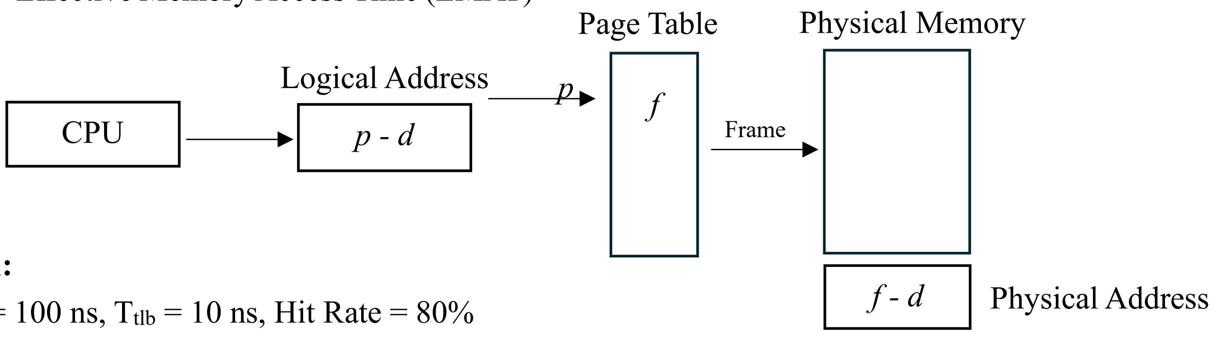
- Difference between Paging and Segmentation

PAGING	SEGMENTATION
✓ Divides physical and logical memory into **fixed-size** units called **frames** and **pages.**	✓ Divides logical memory into **variable-sized units** called **segments** (e.g., code, data, stack).
✓ The program views a single, continuous, linear address space.	✓ The program views a collection of multiple address spaces (one per segment).
✓ **Internal Fragmentation**: The last page of a process is likely to be only partially full.	✓ **External Fragmentation**: Free memory is broken into small, non-contiguous blocks over time, requiring compaction.

- The Translation Lookaside Buffer (TLB) and its impact on performance
  - ✓ The **TLB** is the CPU's address **translation cache**. It speeds up memory access by storing recent address lookups, preventing slow RAM searches for every memory request. This makes virtual memory fast and practical. Performance depends on the TLB's high success rate at finding cached translations.

Layer 1: Classic OS Problems Analysis – Comprehensive Report

- Effective Memory Access Time (EMAT)



Given:

$T_{mem} = 100$ ns, $T_{tlb} = 10$ ns, Hit Rate = 80%

$\text{Time}_{hit} = \text{Ttlb} + \text{Tmem}$

$= 10 + 100 = \mathbf{110\ ns}$

$\text{Time}_{miss} = \text{Ttlb} + 2 \times \text{Tmem}$

$= 10 + 2 \times 100 = \mathbf{210\ ns}$

EMAT FORMULA:

$\text{EMAT} = (\text{Hit rate}) \times \text{Time}_{hit} + (1\text{-Hit rate}) \times \text{Time}_{miss}$

$= 0.8 \times 110 + 0.2 \times 210$

$= 88 + 42$

$\text{EMAT} = \boxed{\mathbf{130\ ns}}$

PART C: CPU Scheduling Principles

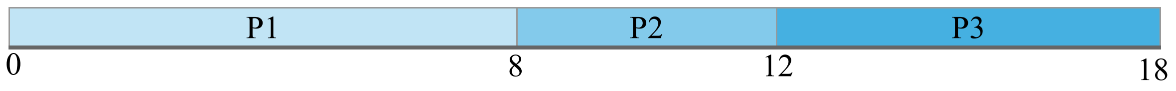
- FCFS vs. Round Robin Scheduling

PROCESS	**THREADS**
✓ The process that requests the CPU first is allocated the CPU first.	✓ Designed for time-sharing systems.
✓ The implementation of the FCFS policy is easily managed with a FIFO queue.	✓ Similar to FCFS scheduling, but preemption is added to switch between processes.
✓ The CPU is allocated to the process at the head of the ready queue when free, while new processes are added to the tail.	✓ A small unit of time, called a time quantum or time slice, is defined (generally from 10 to 100 ms).
✓ The average waiting time is often quite long.	✓ The ready queue is treated as a circular queue.
✓ This algorithm is nonpreemptive and is troublesome for time-sharing systems.	✓ The CPU scheduler cycles through the ready queue, allocating the CPU to each process for one time quantum.

Layer 1: Classic OS Problems Analysis – Comprehensive Report

- Gantt Charts and avg. waiting time for FCFS and Round Robin
  P1 = 8ms, P2 = 4ms, P3 = 6ms, quantum = 3ms

FCFS Scheduling

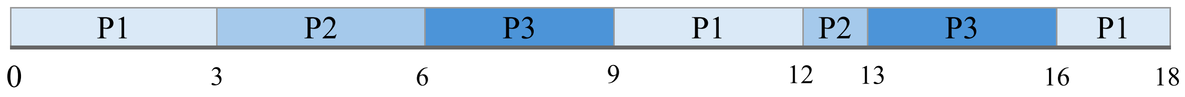


Waiting time for:

P1 = 0 ms
P2 = 8 ms
P3 = 12 ms

Average waiting time = (0 + 8 + 12) / 3 = **6.67 ms**

Round Robin Scheduling



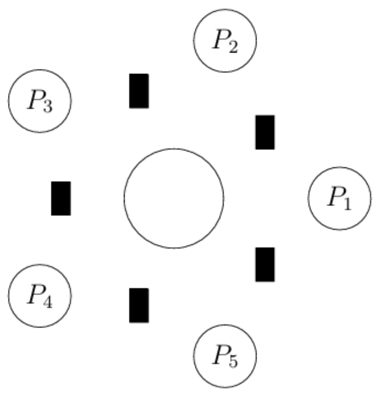
Waiting time for:

P1 = 0 + 6 + 4 = 10 ms
P2 = 3 + 6 = 9 ms
P3 = 6 + 4 = 10 ms

Average waiting time = (10 + 9 + 10) / 3 = **9.67 ms**

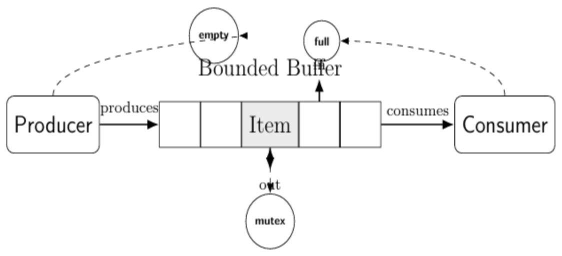
PART D: Classic Synchronization Problems

- Dining Philosophers Problem
  - ✓ The Dining Philosophers problem, formulated by Edsger Dijkstra, is a classic computer science analogy for synchronization issues, describing five philosophers at a circular table who alternate between two states: **thinking** and **eating**. To eat, a philosopher must acquire both their left and right shared forks. This setup introduces two main problems: **deadlock**, which occurs if all philosophers simultaneously grab their left fork and wait indefinitely for their right fork in a circular chain, and **starvation**, where a specific philosopher is perpetually unlucky and prevented from acquiring both forks (e.g., their neighbors always eat first), even if the system as a whole hasn't deadlocked. Solutions focus on breaking this circular wait, such as using a "waiter" (a mutex) to allow only one philosopher to attempt to eat at a time, enforcing a resource



hierarchy (always picking up the lower-numbered fork first), or limiting the number of "hungry" philosophers to four, which ensures at least one can always acquire both forks.

- Producer - Consumer Problem
  - ✓ The Producer-Consumer problem is a classic synchronization issue involving two processes: a "Producer" that adds items to a fixed-size "bounded buffer" and a "Consumer" that removes them. The challenge is to prevent the Producer from adding to a full buffer, the Consumer from removing from an empty buffer, and avoiding race conditions where both access the buffer simultaneously. As the diagram shows, this is solved using three semaphores: an **empty semaphore** (blocking the Producer when the buffer is full), a **full semaphore** (blocking the Consumer when the buffer is empty), and a **mutex semaphore** (ensuring mutual exclusion, so only one process can access the buffer at a time). This ensures safe, deadlock-free, and efficient coordination by allowing processes to sleep when they cannot act and wake precisely when the buffer state changes.



- Reader – Writer Scenarios
  - ✓ The Reader-Writer problem is a classic concurrency scenario where a shared data resource is accessed by two types of processes: "Readers," who can read simultaneously, and "Writers," who must have exclusive access to modify the data. This leads to two main scenarios based on priority: **Readers' Priority**, which allows any number of readers to access the resource as long as one is active, but risks **Writer starvation** if readers are continuous; and **Writers' Priority**, which blocks new readers if a writer is waiting, thus solving writer starvation but risking **Reader starvation**. Because both priority-based models can lead to one process type being indefinitely postponed, balanced solutions, such as a fair **FIFO** queue, are often used to prevent starvation for both.