Great — I'll answer every question in detail for the SPPU TE (2019 pattern) SPOS viva. I'll number answers to match your sheet.

---

1 — Input & output of Pass I and Pass II (Assembler) and Macro processor

Assembler (Two-pass assembler):

- **Pass I (input → output):**
  - **Input:** Assembly source program (lines with labels, mnemonics, operands, directives).
  - **Actions:** Scans program, assigns addresses (LOCCTR), records labels and their addresses in the **Symbol Table (SYMTAB)**, collects literals into Literal Table, processes assembler directives (like START, ORG, EQU), produces an **Intermediate File** (usually with addresses and a form of the statements).
  - **Output:** SYMTAB, Literal Table, Intermediate File (addressed listing).
- **Pass II (input → output):**
  - **Input:** The Intermediate File, SYMTAB, Literal Table.
  - **Actions:** Resolves addresses and symbols, generates machine code/object code for each instruction, resolves literals, creates object program records (text, header, end).
  - **Output:** Final object program (object code), possibly a listing with addresses and object code.

Macro Processor (typical two-phase view):

- **Macro Definition Phase (Pass I for macros):**
  - **Input:** Source containing macro definitions and macro calls.
  - **Actions:** Reads macro definitions, stores macro bodies in **Macro Definition Table (MDT)** and entries in **Macro Name Table (MNT)**, records parameter names in **Argument List Arrays (ALA)**.
  - **Output:** Tables: MNT, MDT, ALA (macro definition information).
- **Macro Expansion Phase (Pass II for macros):**
  - **Input:** Source with macro calls + macro tables.
  - **Actions:** When encountering a macro call, expand it using MDT and ALA — substitute parameters and insert expanded statements into the output stream (or intermediate).
  - **Output:** Expanded source (with macros replaced by their bodies), which then goes to assembler proper.

---

2 — Preemptive vs Non-preemptive Scheduling

- **Non-preemptive (cooperative) scheduling:** Once CPU is assigned to a process, it keeps CPU until it **terminates or blocks** (I/O etc.). The scheduler cannot forcibly remove it.

  - **Examples:** FCFS (First Come First Serve), Non-preemptive SJF, Non-preemptive Priority.
  - **Pros:** Simple, low context switching overhead.
  - **Cons:** Poor responsiveness for short/interactive tasks; long jobs can block others (convoy effect).
- **Preemptive scheduling:** Scheduler may **interrupt** a running process and switch CPU to another (based on time slice, higher priority arrival, etc.).

  - **Examples:** Round Robin, Preemptive SJF (SRTF), Preemptive Priority, Multilevel Feedback Queue.
  - **Pros:** Better responsiveness, supports time-sharing, can meet priority/real-time requirements.
  - **Cons:** More context switching overhead, more complex to implement.

---

3 — Turnaround time, Response time, Waiting time

- **Turnaround Time (TAT):** Total time taken from **submission (arrival)** of a process to its **completion**.
  - Formula: `TAT = Completion time − Arrival time`.
- **Waiting Time (WT):** Total time a process spends **in ready queue** waiting for CPU (excluding execution and I/O).
  - Formula: `WT = Turnaround time − Burst time (CPU time)`.
- **Response Time (RT):** Time from submission until the process produces its **first response / first gets CPU**. For interactive systems RT matters.
  - Formula: `RT = Time of first CPU allocation − Arrival time`.

Example: Process arrives at t=0, starts at t=3, finishes at t=8 (burst=5): RT=3, TAT=8, WT=TAT−burst=3.

---

4 — Which is best when comparing FCFS, RR, Priority, and SJF?

“Best” depends on the metric:

- **SJF (Shortest Job First)** — *Best* for **minimizing average waiting time** (non-preemptive SJF) and hence average turnaround, but can cause **starvation** of long jobs and requires knowledge of burst times (or good estimate).
- **Round Robin (RR)** — *Best* for **time-sharing / fairness / response time** for interactive systems (if quantum chosen well). High quantum → behaves like FCFS; very small quantum → high context switching.
- **Priority Scheduling** — *Best* when tasks have different criticality; can provide quick service to high priority tasks but may **starve** low priority jobs unless aging is used. Can be preemptive or non-preemptive.
- **FCFS** — Simple and fair in arrival order, **poor average waiting time** and poor response for small jobs behind long jobs (convoy effect).
  Conclusion: For throughput/avg waiting time — SJF; for interactive response — RR; for critical tasks — Priority; FCFS is simplest but usually not optimal.

---

5 — Which page replacement algorithm is best when LRU, Optimal, and FIFO are compared?

- **Optimal (Belady’s algorithm):** Replaces the page that **won’t be used for the longest time in the future**. It **minimizes** page faults (theoretical lower bound). **Best** in terms of page faults but **unimplementable** in practice because it needs future knowledge.
- **LRU (Least Recently Used):** Replace page that was **least recently used**. Good practical approximation to optimal, performs well in typical workloads; requires support (stack, counters) or hardware reference bit.
- **FIFO:** Replace the page that came **first** into memory. Simpler but can perform badly (Belady anomaly — more frames can increase faults).
  Conclusion: Optimal is best theoretically; LRU is best practical option in many scenarios; FIFO is simplest but often worst.

---

6 — What is Macro and Assembler?

- **Macro:** A user-defined name representing a sequence of assembly statements (template). Macro allows parameterized code expansion; reduces code repetition. A macro call is replaced by its macro body during macro expansion.
- **Assembler:** A translator that converts assembly language (mnemonics + operands) into machine code/object code. It also handles symbol resolution, addresses, literal processing, and assembler directives.

---

7 — Why is Banker’s algorithm required?

- **Banker’s algorithm** is a **deadlock avoidance** algorithm. It checks system’s state and decides if resource allocation for a request will keep system in a **safe state** (no deadlock risk). It’s required for systems where resources are safe-allocated based on maximum claims — it ensures that granting requests will not lead to a possible deadlock later. Useful in systems with multiple instances of resources and where processes declare maximum needs in advance.

---

8 — What is deadlock? Four necessary conditions

- **Deadlock:** A situation where a set of processes are each waiting for events (usually resources) that can be caused only by other processes in the set — so none can proceed.
- **Four necessary (Coffman) conditions (all must hold):**
    1. **Mutual Exclusion:** At least one resource is non-shareable.
    2. **Hold and Wait:** A process holds at least one resource and is waiting to acquire additional resources held by others.
    3. **No Preemption:** Resources cannot be forcibly taken from a process; they must be released voluntarily.
    4. **Circular Wait:** There exists a cycle of processes `{P0 waits for resource held by P1, P1 waits for resource held by P2, ..., Pn waits for resource held by P0}`.

To prevent deadlocks, the system must ensure at least one of these conditions is impossible.

---

9 — Solve 1 example of each: FCFS, RR, SJF, Priority (with calculations)

Use three processes P1, P2, P3 all arriving at time 0 with CPU bursts:

- P1 = 5, P2 = 3, P3 = 1.

A) FCFS (arrival order P1, P2, P3)

Gantt: `P1 (0–5) | P2 (5–8) | P3 (8–9)`

- Completion times: P1=5, P2=8, P3=9
- Turnaround times: TAT1=5−0=5, TAT2=8−0=8, TAT3=9−0=9
- Waiting times: W1=5−5=0, W2=8−3=5, W3=9−1=8
- **Average TAT = (5+8+9)/3 = 22/3 = 7.33**
- **Average WT = (0+5+8)/3 = 13/3 = 4.33**

B) Round Robin (quantum = 2)

We run time slices of 2 units in arrival order; track remaining bursts.

Initial remaining: P1=5,P2=3,P3=1.

Gantt timeline:

- 0–2: P1 (remains 3)
- 2–4: P2 (remains 1)
- 4–6: P3 (needs 1) — but P3 has burst 1, so runs 4–5 and finishes at 5
  (adjust next slots accordingly)
  Let's simulate carefully:

Step by step:

- 0–2: P1 (rem=3)
- 2–4: P2 (rem=1)
- 4–5: P3 (rem=0) — P3 completes at t=5
- 5–7: P1 (rem=1)
- 7–8: P2 (rem=0) — P2 completes at t=8
- 8–9: P1 (rem=0) — P1 completes at t=9

Completion times: P3=5, P2=8, P1=9
Turnaround: TAT1=9, TAT2=8, TAT3=5
Waiting: W1=9−5=4, W2=8−3=5, W3=5−1=4
Average TAT = (9+8+5)/3 = 22/3 = 7.33
Average WT = (4+5+4)/3 = 13/3 = 4.33

(Observation: with this particular example RR produced same averages as FCFS because of arrival and bursts chosen — but response times differ.)

Response times (first time each got CPU):
P1 first at 0 → RT1=0, P2 first at 2 → RT2=2, P3 first at 4 → RT3=4.

C) SJF (non-preemptive)

Order by shortest burst first (all arrived at 0): P3 (1), P2 (3), P1 (5)

Gantt: `P3 (0–1) | P2 (1–4) | P1 (4–9)`

- Completion times: P3=1, P2=4, P1=9
- TAT: P3=1, P2=4, P1=9
- WT: P3=0, P2=1, P1=4
- **Avg TAT = (1+4+9)/3 = 14/3 = 4.67**
- **Avg WT = (0+1+4)/3 = 5/3 = 1.67**

SJF gives the best average waiting time here.

D) Priority Scheduling (non-preemptive)

Assign priorities (lower = higher priority): P1:2, P2:1, P3:3 (all arrive at 0). Order by priority: P2 (1), P1 (2), P3 (3).

Gantt: `P2 (0–3) | P1 (3–8) | P3 (8–9)`

- Completion times: P2=3, P1=8, P3=9
- TAT: P2=3, P1=8, P3=9
- WT: P2=0, P1=3, P3=8
- Avg TAT = (3+8+9)/3 = 20/3 = 6.67
- Avg WT = (0+3+8)/3 = 11/3 = 3.67

---

10 — Examples: FIFO, Optimal, LRU page replacement

Use reference string: `7, 0, 1, 2, 0, 3, 0, 4, 2` with **3 frames**. Start frames empty.

A) FIFO (3 frames)

We insert in arrival order, replace oldest.

Step through:

1. 7 — miss → frames [7, -, -] (faults=1)
2. 0 — miss → [7,0,-] (2)
3. 1 — miss → [7,0,1] (3)
4. 2 — miss → replace 7 → [2,0,1] (4)
5. 0 — hit → [2,0,1]
6. 3 — miss → replace 0 → [2,3,1] (5)
7. 0 — miss → replace 1 → [2,3,0] (6)
8. 4 — miss → replace 2 → [4,3,0] (7)
9. 2 — miss → replace 3 → [4,2,0] (8)

FIFO page faults = **8**.

B) Optimal (3 frames)

Replace page not used for longest time in future.

Sequence:

1. 7 — miss → [7,-,-] (1)
2. 0 — miss → [7,0,-] (2)
3. 1 — miss → [7,0,1] (3)
4. 2 — miss → look ahead: usages: 7(not used again),0 used at position 5,1 not used again → replace 7 or 1 (both not used again). Choose one (commonly 7). → [2,0,1] (4)
5. 0 — hit
6. 3 — miss → future uses: 2 used at last step (pos 9),0 used later? 0 not used again after pos5? Check sequence positions: after 3 we have `0,4,2` so 0 used at next step (pos7), 1 not used again → replace 1 → [2,0,3] (5)
7. 0 — hit
8. 4 — miss → look ahead: 2 used later (pos9),0 not used again after pos7 → replace 0 → [2,4,3] (6)
9. 2 — hit

Optimal page faults = **6**.

C) LRU (3 frames)

Replace page used least recently.

Simulate:

1. 7 — miss → [7,-,-] (1)
2. 0 — miss → [7,0,-] (2)
3. 1 — miss → [7,0,1] (3)
4. 2 — miss → least recently used among {7,0,1} — usages before step4: 7 at step1,0 at step2,1 at step3 → LRU=7 → replace 7 → [2,0,1] (4)
5. 0 — hit (update recency of 0)
6. 3 — miss → recencies: 2 used at step4,0 at step5,1 at step3 → LRU=1 → replace 1 → [2,0,3] (5)
7. 0 — hit
8. 4 — miss → recencies: 2 used at step4,0 at step7,3 at step6 → LRU=2 → replace 2 → [4,0,3] (6)
9. 2 — miss → recencies: 4 used at step8,0 at step7,3 at step6 → LRU=3 → replace 3 → [4,0,2] (7)

LRU page faults = 7.

Summary: FIFO=8, Optimal=6 (best), LRU=7.

---

11 — Difference between Pass I and Pass II of assembler

- **Pass I:** Build symbol table (labels → addresses), compute addresses (LOCCTR), collect literals, check syntax to intermediate level, produce intermediate file. Cannot produce final object code because forward references unknown.
- **Pass II:** Use intermediate file + SYMTAB to generate final object code, resolve references, generate object program records.
  (See Q1 for inputs/outputs.)

---

12 — Drawback of Pass I of assembler

- **Primary drawback:** Pass I **does not generate final object code**; forward references cannot be resolved in pass I, so a second pass is required. Also, if assembler is single-pass designed, pass I may not detect some semantic errors that are only resolvable with address resolution. In short: **incomplete code generation — needs pass II**.

---

13 — Difference between Compilers and Interpreters

- **Compiler:** Translates entire high-level program into machine/assembly/object code **before execution** (creates an executable).
  - **Pros:** Faster execution (compiled code), optimizations possible, single translation step.
  - **Cons:** Longer edit-compile-run cycles, platform dependent (binary).
  - **Examples:** C, C++ compilers.
- **Interpreter:** Translates and **executes line-by-line** or statement-by-statement at runtime. No standalone binary produced.
  - **Pros:** Good for rapid development and debugging, platform independence if interpreter available.
  - **Cons:** Slower execution (translation at runtime), must have interpreter present.
  - **Examples:** Python, Ruby (traditional interpreters).
- **Hybrid approaches:** Bytecode + virtual machine (Java: compile to bytecode, then interpreted/JIT), Just-In-Time (JIT) compilers.

---

14 — Different parts an assembly language is divided into

Typical assembly program has these logical parts:

1. **Header / Program start directive** (e.g., START, ORG) — sets starting address.
2. **Data section / Data directives** — constants, variables, storage allocation (`DB`, `DW`, `RESW`, etc.).
3. **Code / Text section** — sequence of instructions (labels, mnemonics, operands).

4. **Macros / Macro definitions** (optional) — macro name and body.
5. **Assembler directives / pseudo-ops** — directives for assembler (EQU, END, LTORG, INCLUDE).
6. **Literal pool / literals** — constants referenced as literals.
7. **End directive** — END, entry point.
   You can also mention **symbol table**, **literal table**, and **comment fields** (for readability).

---

15 — What is a process? Different states of processes

- **Process:** A program in execution — it includes program code, current activity (program counter, registers), stack, data section and other OS-maintained information (PCB). Processes are dynamic instances of programs.

Common process states:

1. **New:** Being created.
2. **Ready:** Loaded and waiting in ready queue for CPU allocation.
3. **Running:** CPU is executing the process.
4. **Waiting (Blocked):** Waiting for I/O or some event/resource.
5. **Terminated (Exit):** Finished execution or killed.
   Additionally some systems have:

- **Suspended (Ready Suspended / Blocked Suspended):** Swapped out from main memory (for mid-term swapping).

State transitions: e.g., New → Ready → Running → Waiting → Ready → Running → Terminated.

---

16 — What is Paging?

- **Paging:** Memory management scheme that divides logical (virtual) address space into fixed-size blocks called **pages** and physical memory into fixed-size blocks called **frames** (same size). The OS maps pages to frames using a **page table**. Logical addresses split into **page number** and **offset**. Paging eliminates external fragmentation and simplifies memory allocation; supports virtual memory (demand paging).
- **Drawbacks:** Internal fragmentation (within a page), overhead for page tables (multi-level page tables, TLB used to speed lookups).

---

17 — What is Segmentation?

- **Segmentation:** Memory management where a program is divided into variable-sized logical units called **segments** (e.g., code, data, stack, modules). Each segment has a **segment number** and an **offset** within that segment. The OS uses a **segment table** mapping segment numbers to base addresses and lengths.
- **Benefits:** Matches programmer's logical view, easier to implement sharing and protection at module granularity, no internal fragmentation inside a segment but external fragmentation possible. Can be combined with paging (paged segmentation) to reduce fragmentation.

---

18 — What are Pages and Frames?

- **Pages:** Fixed-size blocks of the **logical/virtual address space**. Page size common values: 4KB, etc.
- **Frames:** Fixed-size blocks of **physical memory (RAM)** of same size as pages. A page is loaded into a frame when resident. Pages map to frames via page table.

---

19 — Why is there need of CPU scheduling algorithms?

CPU scheduling algorithms are required to:

- **Maximize CPU utilization** (keep CPU busy).
- **Maximize throughput** (processes completed per unit time).

- **Minimize turnaround time, waiting time, and response time** depending on system goals.
- **Provide fairness** among processes and meet priority or real-time constraints.
- **Support multiprogramming and interactive use** (short response times for users).
  Different algorithms trade off these goals (throughput vs fairness vs response).

---

20 — Different types of schedulers

- **Long-term scheduler (Job scheduler):** Controls degree of multiprogramming by deciding which processes/jobs are admitted to the system (from job pool to ready queue). Runs relatively infrequently.
- **Mid-term scheduler (Swapper):** Temporarily removes processes from memory (suspends/swaps out) and later reintroduces them — used for memory management and balancing. Runs occasionally.
- **Short-term scheduler (CPU scheduler):** Selects which ready process will execute next on CPU. Runs very frequently (milliseconds) — performance critical.

---

21 — Explain long-term, middle-term and short-term scheduler (details)

- **Long-term scheduler (job scheduler):**
    - **Function:** Admit jobs into the system (decides which jobs are loaded into memory for execution).
    - **Frequency:** Infrequent (seconds or more).
    - **Goal:** Control job mix and degree of multiprogramming (I/O-bound vs CPU-bound mix).
    - **Typical in batch systems.**
- **Middle-term scheduler (swapper, intermediate):**
    - **Function:** Swap processes in/out of memory to control memory usage and improve process mix. May suspend processes to disk and later resume.
    - **Frequency:** Moderate (on memory pressure or policy triggers).
    - **Goal:** Improve memory utilization, reduce page faults, manage multiprogramming level dynamically.
- **Short-term scheduler (CPU scheduler):**
    - **Function:** Choose among ready processes which one gets CPU next.
    - **Frequency:** Very high (on each scheduling event: timer interrupt, I/O completion, process termination).
    - **Goal:** Optimize responsiveness, throughput, fairness, depending on chosen algorithm (RR, SJF, Priority, etc.).

If you want, I can:

- Create **one-page concise revision notes** from the above for your viva (bullet points and key formulas).
- Or prepare a few **short viva Q-A flashcards** with one-line answers for quick practice.

Which follow-up would help you most?