CS 440
Programming Assignment 1
Process Control Block (PCB) & State Transition Simulator
Due in Canvas by Sunday, February 15, 2026 @ 11:59 PM

Assignment Overview

In this assignment, you will build a small operating system process simulator. Your program will not create real operating system processes. Instead, it will simulate how an operating system manages processes internally using Process Control Blocks (PCBs), process states, and queues. You can work individually or in pairs. This assignment aligns with Chapters 1–3 of Silberschatz, Operating System Concepts (10th Edition).

Allowed Programming Languages

You may complete this assignment using one of the following languages:
- C
- C++
- Java
- C#
- Python

Starter templates are provided for Java, Python, and C.

Learning Objectives

- Implement Process Control Blocks (PCBs)
- Correctly model process states and transitions
- Use queues and tables to simulate OS data structures
- Detect and report illegal process state transitions
- Demonstrate understanding through logs and STATUS snapshots

Process States

Each process must always be in exactly one of the following states:
NEW, READY, RUNNING, WAITING, TERMINATED

At most one process may be in the RUNNING state at any time.

Process Control Block (PCB)

Each process must have a PCB containing at least:
- Process ID (pid)
- Process name (unique)
- Current state
- Priority (integer)
- Program counter (pc, simulated)

- Total CPU time used (cpuTime, simulated)

You may include additional fields if helpful.

Supported Commands (Simplified)

Your program reads a trace file (plain text). Each non-empty, non-comment line is one command.
Lines beginning with # are comments.

Commands used in the trace file:

```
CREATE <name> <priority>
DISPATCH
TICK <n>
BLOCK <name>
WAKE <name>
EXIT <name>
STATUS
```

Extra credit command:

```
KILL <name>
```

Command Meanings

- **CREATE**
  Create a new process and place it into READY
  NEW → READY
- **DISPATCH**
  If the CPU is idle, move the next READY process to RUNNING
  READY → RUNNING
  (FIFO scheduling, no preemption)
- **TICK n**
  The RUNNING process uses n CPU ticks
  pc += n, cpuTime += n
- **BLOCK**
  The RUNNING process blocks for I/O
  RUNNING → WAITING
- **WAKE**
  A WAITING process finishes I/O
  WAITING → READY

- **EXIT**
  The RUNNING process finishes execution
  RUNNING → TERMINATED

- **STATUS**
  Print a snapshot of the current system state

- **KILL <name>**

  Immediately terminate the specified process, regardless of its state.
  If the process is in READY: remove it from the READY queue
  READY → TERMINATED
  If the process is in WAITING: remove it from the WAITING queue
  WAITING → TERMINATED
  If the process is RUNNING: stop it immediately and free the CPU
  RUNNING → TERMINATED (CPU becomes idle)
  If the process does not exist or is already TERMINATED: this is an ERROR
  The PCB for the process remains in the process table with state TERMINATED, so it can still appear in the STATUS output.

Scheduling Rule

FIFO scheduling is required:
- DISPATCH selects the front of the READY queue
- No preemption is allowed
- DISPATCH is illegal if a process is already RUNNING

Exact Required Output Format

For each command, print exactly one log line:

```
[step=6] CMD=BLOCK P1 | OK | P1: RUNNING -> WAITING
```

For errors:

```
[step=10] CMD=DISPATCH | ERROR | CPU already running P2
```

When STATUS is printed, use this format:

```
RUNNING: <process name or NONE>
READY: [P1, P3]
WAITING: [P2]

TABLE:
PID  NAME  STATE       PRIO  PC  CPUTIME
1    P1    READY       3     5   5
2    P2    WAITING     1     2   2
----
```

TABLE Column Meanings

- **PID**
  The **Process ID**, a unique number assigned to each process when it is created. The OS uses the PID to identify and track processes internally.
- **NAME**
  The **process name** given in the trace file (e.g., P1, P2). Names are unique and are used in commands such as BLOCK, WAKE, EXIT, and KILL.
- **STATE**
  The **current state** of the process. This will be one of:
  NEW, READY, RUNNING, WAITING, or TERMINATED.
- **PRIO**
  The **priority value** assigned when the process was created. In this assignment, priority is stored in the PCB but **does not affect scheduling**, which is FIFO.
- **PC**
  The **program counter**, a simulated value representing how much of the process has executed. This value increases only when the process is RUNNING and a TICK command occurs.
- **CPUTIME**
  The **total CPU time** the process has used so far. This value is the sum of all CPU ticks assigned to the process across its lifetime.

BearID Auto-STATUS Rule

Let N be the last digit of the last four digits of your UNC BearID.
Your program must automatically print STATUS every (N + 3) steps.

Example: BearID last 4 digits 1234 → N=4 → auto STATUS every 7 steps.

At program startup, print:

```
BearID last digit: 4
Auto STATUS every: 7 steps
```

Pair Rule

Pairs (2 students max) are allowed.
- Each student computes their own N value
- Use N = max(N1, N2)
- Auto STATUS interval = N + 3

Both students' names and last 4 of their BearIDs must appear in the code header comment and README.

Extra Credit: KILL <name> Command (Up to 4 Points)

Implement a KILL <name> command that immediately terminates a process from any state.

Important: KILL is handled exactly like other commands:
- It appears as a line in the trace file (e.g., "KILL P2")
- Your command parser reads it and dispatches to a handleKill(...) function (e.g., switch/case)
- It must produce exactly one log line (OK or ERROR)

What KILL must do:
1) If the named process does not exist → log ERROR and do nothing.
2) If the named process is already TERMINATED → log ERROR (already terminated) and do nothing.
3) Otherwise, force the process to TERMINATED and clean up OS structures:
  - If in READY: remove it from the READY queue.
  - If in WAITING: remove it from the WAITING queue.
  - If RUNNING: set running process to NONE (CPU becomes idle).
  - If NEW (if you ever keep NEW around): terminate it.

KILL does NOT remove the PCB from the process table. The PCB remains in the table with state TERMINATED
(so we can still see it in STATUS / final tables).

KILL does not change any other process's state except freeing the CPU if the killed process was RUNNING.

Extra Credit Scoring (Up to +4)

+2 points: KILL correctly terminates processes from READY, WAITING, and RUNNING, and removes them from the correct queue(s)
+1 point: KILL is demonstrated correctly using the provided kill test trace (see Example 2 below)
+1 point: Robust edge-case handling (e.g., repeated KILL, killing last remaining process, KILL followed by DISPATCH behaves correctly)

Examples (Full Input and Full Expected Output)

Below are examples showing exactly what your program output should look like. Your numbers (PIDs, PCs, cpuTime) must be consistent with your implementation and the trace.

Example 1: Lifecycle + One ERROR (No KILL)

Input trace (example1_trace.txt):

```
CREATE P1 3
CREATE P2 1
DISPATCH
TICK 4
BLOCK P1
DISPATCH
DISPATCH
```

```
TICK 2
WAKE P1
EXIT P2
DISPATCH
TICK 3
EXIT P1
STATUS
```

Expected output (example1_output.txt):

```
BearID last digit: 4
Auto STATUS every: 7 steps


---- BEGIN LOG ----
[step=1] CMD=CREATE P1 3 | OK | P1: NEW -> READY (pid=1)
[step=2] CMD=CREATE P2 1 | OK | P2: NEW -> READY (pid=2)
[step=3] CMD=DISPATCH | OK | P1: READY -> RUNNING
[step=4] CMD=TICK 4 | OK | P1: pc+=4 cpuTime+=4
[step=5] CMD=BLOCK P1 | OK | P1: RUNNING -> WAITING
[step=6] CMD=DISPATCH | OK | P2: READY -> RUNNING
[step=7] CMD=DISPATCH | ERROR | CPU already running P2
RUNNING: P2
READY: []
WAITING: [P1]

TABLE:
PID  NAME  STATE       PRIO  PC  CPUTIME
1    P1    WAITING     3     4   4
2    P2    RUNNING     1     0   0

[step=8] CMD=TICK 2 | OK | P2: pc+=2 cpuTime+=2
[step=9] CMD=WAKE P1 | OK | P1: WAITING -> READY
[step=10] CMD=EXIT P2 | OK | P2: RUNNING -> TERMINATED
[step=11] CMD=DISPATCH | OK | P1: READY -> RUNNING
[step=12] CMD=TICK 3 | OK | P1: pc+=3 cpuTime+=3
[step=13] CMD=EXIT P1 | OK | P1: RUNNING -> TERMINATED
[step=14] CMD=STATUS | OK
RUNNING: NONE
READY: []
WAITING: []

TABLE:
PID  NAME  STATE       PRIO  PC  CPUTIME
1    P1    TERMINATED  3     7   7
```

```
2    P2    TERMINATED  1     2   2
---- END LOG ----
```

Example 2: KILL Extra Credit (Terminate from READY/WAITING/RUNNING)

Input trace (example2_kill_trace.txt):

```
CREATE P1 3
CREATE P2 1
CREATE P3 2
DISPATCH
TICK 1
BLOCK P1
DISPATCH
TICK 1
KILL P3
KILL P1
KILL P2
DISPATCH
STATUS
```

Expected output (example2_kill_output.txt):

```
BearID last digit: 0
Auto STATUS every: 3 steps


---- BEGIN LOG ----
[step=1] CMD=CREATE P1 3 | OK | P1: NEW -> READY (pid=1)
[step=2] CMD=CREATE P2 1 | OK | P2: NEW -> READY (pid=2)
[step=3] CMD=CREATE P3 2 | OK | P3: NEW -> READY (pid=3)
RUNNING: NONE
READY: [P1, P2, P3]
WAITING: []

TABLE:
PID  NAME  STATE       PRIO  PC  CPUTIME
1    P1    READY       3     0   0
2    P2    READY       1     0   0
3    P3    READY       2     0   0

[step=4] CMD=DISPATCH | OK | P1: READY -> RUNNING
[step=5] CMD=TICK 1 | OK | P1: pc+=1 cpuTime+=1
[step=6] CMD=BLOCK P1 | OK | P1: RUNNING -> WAITING
RUNNING: NONE
```

```
READY: [P2, P3]
WAITING: [P1]

TABLE:
PID  NAME  STATE       PRIO  PC  CPUTIME
1    P1    WAITING     3     1   1
2    P2    READY       1     0   0
3    P3    READY       2     0   0

[step=7] CMD=DISPATCH | OK | P2: READY -> RUNNING
[step=8] CMD=TICK 1 | OK | P2: pc+=1 cpuTime+=1
[step=9] CMD=KILL P3 | OK | P3: READY -> TERMINATED (removed from
READY)
RUNNING: P2
READY: []
WAITING: [P1]

TABLE:
PID  NAME  STATE       PRIO  PC  CPUTIME
1    P1    WAITING     3     1   1
2    P2    RUNNING     1     1   1
3    P3    TERMINATED  2     0   0

[step=10] CMD=KILL P1 | OK | P1: WAITING -> TERMINATED (removed
from WAITING)
[step=11] CMD=KILL P2 | OK | P2: RUNNING -> TERMINATED (CPU now
NONE)
[step=12] CMD=DISPATCH | ERROR | no READY processes
RUNNING: NONE
READY: []
WAITING: []

TABLE:
PID  NAME  STATE       PRIO  PC  CPUTIME
1    P1    TERMINATED  3     1   1
2    P2    TERMINATED  1     1   1
3    P3    TERMINATED  2     0   0

[step=13] CMD=STATUS | OK
RUNNING: NONE
READY: []
WAITING: []

TABLE:
PID  NAME  STATE       PRIO  PC  CPUTIME
```

```
1     P1     TERMINATED  3      1    1
2     P2     TERMINATED  1      1    1
3     P3     TERMINATED  2      0    0
---- END LOG ----
```

How to Test Your Program (Checklist)

1) Pick your BearID last digit N (last digit of the last four digits).
2) Compute R = N + 3.
3) Run trace1_happy.txt and confirm:
  - No crashes
  - Transitions follow the state diagram
  - READY/WAITING contains the correct processes in the correct order
  - pc and cpuTime increase only on TICK
4) Run trace2_errors.txt and confirm:
  - ERROR lines appear when commands are illegal
  - Program continues running after errors
5) (Extra credit) Implement KILL and run trace3_kill_extra.txt:
  - KILL removes process from READY/WAITING queues as appropriate
  - Killing RUNNING makes RUNNING become NONE
  - Killed PCBs remain in TABLE as TERMINATED

Rubric (40 points + up to 4 extra credit)

Compiles & Runs (5 pts)

- Excellent (5): Compiles and runs for all required traces without crashing.
- Good (4): Runs but has minor issues on one trace (e.g., formatting or one edge-case).
- Needs Work (2): Compiles but crashes or fails on multiple traces.
- Missing (0): Does not compile or cannot be executed.

PCB Implementation (6 pts)

- Excellent (6): All required PCB fields present and updated correctly.
- Good (5): All fields present; one field is inconsistent or not used correctly.
- Needs Work (3): Multiple fields are missing or frequently incorrect.
- Missing (0): PCB structure missing or unusable.

Process Table Management (4 pts)

- Excellent (4): Process table stays consistent; lookup by name works reliably; terminated processes remain visible.
- Good (3): Mostly consistent; minor lookup or table inconsistencies.
- Needs Work (2): Table is frequently inconsistent or loses processes.

- Missing (0): No process table or unusable table.

READY / WAITING Queues (6 pts)

- Excellent (6): Queues always correct; FIFO order preserved; no duplicates.
- Good (5): Mostly correct; minor ordering or duplicate edge-case issues.
- Needs Work (3): Incorrect queue behavior in multiple places.
- Missing (0): Queues not implemented or incorrect throughout.

Dispatch & CPU Rules (5 pts)

- Excellent (5): Single RUNNING invariant always enforced; DISPATCH errors when CPU is busy; correct handling when READY is empty.
- Good (4): One minor invariant or edge-case issue.
- Needs Work (2): Multiple violations of CPU/RUNNING rules.
- Missing (0): Dispatch logic missing or fundamentally incorrect.

State Transitions (8 pts)

- Excellent (8): All legal transitions implemented correctly *(CREATE / DISPATCH / TICK / BLOCK / WAKE / EXIT)*.
- Good (6): One or two transition mistakes but overall correct behavior.
- Needs Work (3): Frequent transition mistakes or incorrect state diagram logic.
- Missing (0): State transitions not implemented.

Error Handling (3 pts)

- Excellent (3): Illegal commands produce clear, informative ERROR lines and the program continues executing.
- Good (2): Errors detected, but message clarity or consistency is weak.
- Needs Work (1): Some illegal commands not detected; occasional crashes.
- Missing (0): No error handling; crashes or silent failures.

Logging & STATUS Output (Content + Formatting) (3 pts)

- Excellent (3):
    - One log line per command
    - STATUS includes RUNNING, READY, WAITING, and TABLE
    - STATUS table columns are aligned/justified and readable
- Good (2):
    - All required information present
    - Minor spacing or alignment issues, but table is still readable

- Needs Work (1):
    - STATUS present but poorly formatted or difficult to read
- Missing (0):
    - Missing STATUS sections or no meaningful formatting

BearID Auto-STATUS Rule (3 pts)

- Excellent (3): Correct N and R printed at startup; auto STATUS printed exactly every R steps.
- Good (2): Correct computation, but auto STATUS occasionally missing or extra.
- Needs Work (1): Incorrect computation or inconsistent auto STATUS behavior.
- Missing (0): No BearID-based auto STATUS.

README Explanation (4 pts)

- Excellent (4): All required questions answered clearly and tied directly to the student's own output files.
- Good (3): All questions answered, but lacks specificity or output references.
- Needs Work (2): Some questions missing or vague.
- Missing (0): No README or not responsive to prompts.

Extra Credit (Up to +4 points)

Extra credit is added on top of the 40-point base score.

- +2: KILL works correctly for READY, WAITING, and RUNNING and cleans up queues and CPU state
- +1: KILL demonstrated correctly using trace3_kill_extra.txt
- +1: Robust edge-case handling
  (repeated KILL, killing last process, KILL followed by DISPATCH, etc.)
-

Instructor Note

For STATUS output, columns do not need to be fixed-width, but values must line up vertically so the table is readable without guessing which value belongs to which column.

What to Turn In (Submission Requirements)

Submit **one ZIP file** to Canvas that contains last names of participants, e.g., *SmithCS440HW1.zip* or if a pair, *ValdezRichardsCS440HW1.zip,* containing the following items:

1. Source Code

- Your program source files

    - .c, .cpp, .java, .cs, or .py
- The code **must compile and run from the command line**
- Include a **comment header** at the top of your main source file with:
    - Course: **CS 440**
    - Your name(s)
    - Your BearID(s)
    - Date

---

2. Output Files

For each required trace, include the **captured program output**:

- output1.txt – output from trace1_happy.txt
- output2.txt – output from trace2_errors.txt
- output3.txt – output from trace3_kill_extra.txt (extra credit only)

It may be helpful to use output redirection, for example:

./your_program trace1_happy.txt > output1.txt

---

3. README.txt

Include a plain-text file named **README.txt** that answers the following:

1. **BearID Auto-STATUS Rule**
    - What is your BearID last digit?
    - What is your auto-STATUS interval (N + 3)?
2. **Error Handling**
    - Copy **one ERROR line** from output2.txt
    - Explain why that command was illegal
3. **Process Lifecycle**
    - Choose one process from output1.txt

- Describe its state transitions in order

4. **(Extra Credit only)**
    - Briefly explain how your KILL command works and where it removes the process from