Processes

- Process: an abstraction of a running program.

- All runnable software is organized into a number of sequential processes.

- Each process has its own flow of control (i.e. program counter, registers and variables).

- In multiprogramming environment, processes switch back and forth.

- No built-in assumption about timing in programs (use interrupts instead).
Process states

- Three state model:
  1. Running: using CPU currently.
  2. Blocked: unable to run until some external event (e.g. I/O completes) happens.
  3. Ready: runnable, temporarily stopped to let another process run.

- Four possible transitions:
  1. Process blocked for input.
  2. Scheduler picks another process.
  3. Scheduler picks this process.
  4. Input becomes available.
Event wait → Running → Time out

Blocked → Schedule → Ready

Event occurs
- What happens when all processes in memory are blocked for I/O?

- Swapping: move a process from memory to disk (suspend) and bring another process on disk to memory.

- Five state model, add two more states:
  1. Blocked/Suspend: the process is in secondary memory and waiting an event.
  2. Ready/Suspend: the process is in secondary memory but is available for execution as soon as it is loaded into memory.

- Transitions among states:
- A computer system $= n$ sequential processes
  + a scheduler.
  Scheduler does scheduling, handles interrupts and transmits messages between processes.
Interprocess communication (IPC)

- Communication through shared memory.
- Spooling: Simultaneous Peripheral Operation On Line.
- Possible problems (race condition).
  Example: a print spooler.
  - To print a file, a process enters the file name in a designated Spooler directory (an array implemented with circular queue).
  - Another process, printer daemon, prints the files and removes them from the directory.
  - Shared variables: in, out.
  - Print procedure:
    1. in $\rightarrow$ next-free-slot (local variable)
    2. Put the file name to print in array location indexed by next-free-slot
    3. Increment next-free-slot
    4. next-free-slot $\rightarrow$ in.
– Assume two processes A & B and processes can be switched out during executing.
A sequence of actions which can cause problems:

1. Process A: \( in(= 8) \rightarrow next-free-slot \)
2. A is switched out and B is running.
3. Process B: \( in(=8) \rightarrow next-free-slot \)
4. B puts file name to print in Slot 8.
5. B increments local var. to 9.
6. B stores 9 into \( in \).
7. A is scheduled to run again.
8. A puts file name to print into Slot 8.
10. A stores 9 into \( in \).

• Race condition: several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular access order.
Another example.
Bookkeeping application. Need to maintain data coherence, i.e. keep \( a = b \).

**Process 1:** \( a = a + 1; \ b = b + 1 \)

**Process 2:** \( b = 2 \ast b; \ a = 2 \ast a \)

Initially \( a = b \)

**Execution sequence:**

\[
\begin{align*}
    a & = a + 1 \\
    b & = 2 \ast b \\
    b & = b + 1 \\
    a & = 2 \ast a
\end{align*}
\]

At the end \( a \neq b \).

- Critical section: portion of program access shared variable.
- Mutual exclusion: mechanism which makes sure two or more processes do not access a common resource at the same time.
Four conditions to hold to have a good solution for mutual exclusion in critical section.

1. No two processes simultaneously inside their critical sections.

2. No assumptions about relative processor speeds or number of CPU’s.

3. No process stopped outside its critical section should block other processes to enter its critical section.

4. No process should wait arbitrarily long to enter its critical section.
Review:

- Internal structure of O.S.
  - Monolithic systems
  - Layered systems
  - Virtual machines
  - Client-server model

- Process
  - Multiprogramming
  - Process states:
    * Three state model
    * Five state model
Tentative solutions

- Hardware solution:

  1. Disable interrupts
  2. Enter critical section
  3. Do something in critical section
  4. Exit critical section
  5. Re-enable interrupts
- Lock variables

  - A binary shared variable \( lock \).
    \( lock = 1 \): critical region occupied
    \( lock = 0 \): critical region unoccupied

  - The code for entering critical section:

    1. \text{loop: if } lock == 1 \text{ then goto loop;}
    2. \text{lock = 1;}
    3. \text{critical-section();}
    4. \text{lock =0;}

- A possible execution sequence:

1. Process A executes (1) and finds $lock = 0$. Drops from loop.

2. Process A is switched out.

3. Process B checks lock and sees $lock = 0$ and drops from loop.

4. Process B sets $lock = 1$ and enters critical section.

5. Process A wakes up, sets $lock = 1$ (again) and enter critical section.
- **Strict alternation:**

  - Processes take turns to enter critical section.

- **Use a variable** `turn`:
  
  - `turn = 0`: process 0 can enter critical section
  - `turn = 1`: process 1 can enter critical section

- Limitation.
• Peterson’s solution
  
  – Combine lock and take turns.

  – Four possibilities for condition:
    (turn=process && interested[other]=true)
    from the point of view of process 0.

    Case 1: turn = 0, interested[1] = false
    Process 1 is not in critical region.
    Process 0 enters critical region.

    Case 2: turn = 0, interested[1] = true
    Process 1 is in critical region.
    Process 0 waits.

    Case 3: turn = 1, interested[1]=false
    Impossible.

    Case 4: turn = 1, interested[1]=true
    Process 1 is trying to enter critical region, but process 0’s turn first. Process 0 enters critical region.
• Test and Set Lock (TSL) instruction

  – Need hardware support (machine must have this special instruction)

  – TSL: combine
    (Mem) → R and 1 → Mem
    into an atomic operation.
• Sleep and wakeup

  – Sleep: a system call that causes the caller to block until another process wakes it up.

  – Wakeup(p): wake up process p.

  – How to handle wakeup if sent to a process not asleep:

    * Ignore

    * Queue
• Producer consumer problem:

Buffer has $n$ slots. Producer puts item into buffer. Consumer takes item out of buffer.

• Use sleep and wakeup to write procedures for producer and consumer.
● Problem: wakeup sent to process that has not gone to sleep.

● Example:
  – Buffer empty.
  – Consumer reads count==0 and switched out (not sleep yet).
  – Producer enters item in buffer and increments count.
  – Producer sends wakeup.
  – Wakeup lost.
  – Consumer is scheduled to run again.
  – Consumer goes to sleep.
  – Producer eventually fills buffer and goes to sleep.

● Quick fix:
  – Set wakeup waiting bit if wakeup is sent to a non-sleeping process.
  – If a process tries to go sleep and the bit is on, clears the bit and stays awake.

● More than one wakeup?