**Load Control**

**Fundamental tradeoff**

- High multiprogramming level
  - \( MPL_{\text{max}} = \frac{\text{number of page frames}}{\text{minimum number of frames required for a process to execute}} \)

- Low paging overhead
  - \( MPL_{\text{min}} = 1 \) process

- Issues
  - What criterion should be used to determine when to increase or decrease the MPL?
  - Which task should be swapped out if the MPL must be reduced?

**Load Control**

**Thrashing**

- Thrashing can be ameliorated by *local* page replacement

- Better criteria for load control: Adjust MPL so that:
  - \( \text{mean time between page faults (MTBF)} \times \text{page fault service time (PFST)} \)
  - \( \sum WS_i = \text{size of memory} \)

- When the multiprogramming level should be decreased, which process should be swapped out?
  - Lowest priority process?
  - Smallest process?
  - Largest process?
  - Oldest process?
  - Faulting process?

---

**Load Control**

**How not to do it: Base load control on CPU utilization**

- Assume memory is nearly full
- A chain of page faults occur
  - A queue of processes forms at the paging device
- CPU utilization falls
- Operating system increases MPL
  - New processes fault, taking memory away from existing processes
- CPU utilization goes to 0, the OS increases the MPL further...

*System is thrashing* — spending all of its time paging
Saving the world before bedtime

Two Generals’ Problem

Problem:
Design a protocol to save Western Civilization (i.e., one that ensures that Romans always attack simultaneously)

- only communication is by messenger
- messengers must sneak through the valley
- they don’t always make it

Claim: There is no non-trivial protocol that guarantees that the Romans will always attack simultaneously

Proof: Let $P$ be shortest such protocol
- consider last message $m_{\text{last}}$
- $P$ must work if $m_{\text{last}}$ never arrives
- so don’t send it
- but now we have a new protocol shorter than $P$!

Fundamental Limitation: Solution needs
- unbounded number of messages or
- guaranteed message delivery

Otherwise, attack may never take place!
Atomic Commit

The objective

Preserve data consistency for distributed transactions in the presence of failures

Model

- For each distributed transaction $T$:
  - one coordinator
  - a set of participants
- Coordinator knows participants; participants don't necessarily know each other
- Each process has access to a Distributed Transaction Log (DT Log) on stable storage

The setup

- Each process $p_i$ has an input value $\text{vote}_i$:
  $\text{vote}_i \in \{\text{Yes, No}\}$
- Each process $p_i$ has output value $\text{decision}_i$:
  $\text{decision}_i \in \{\text{Commit, Abort}\}$