Processes and Threads

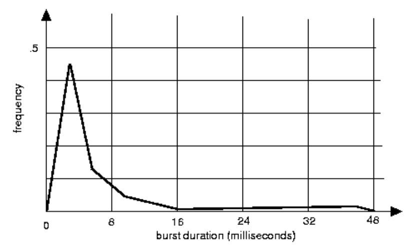
- Processes and their scheduling
- Multiprocessor scheduling
- Threads
- Distributed Scheduling/migration

Processes: Review

- Multiprogramming versus multiprocessing
- Kernel data structure: process control block (PCB)
- Each process has an address space
  - Contains code, global and local variables..
- Process state transitions
- Uniprocessor scheduling algorithms
  - Round-robin, shortest job first, FIFO, lottery scheduling, EDF
- Performance metrics: throughput, CPU utilization, turnaround time, response time, fairness

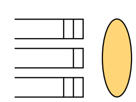
Process Behavior

- Processes: alternate between CPU and I/O
- CPU bursts
  – Most bursts are short, a few are very long (high variance)
  – Modeled using *hyperexponential* behavior
  – If *X* is an *exponential* r.v.
    - $Pr [X <= x] = 1 – e^{-\mu x}$
    - $E[X] = 1/\mu$
  – If *X* is a *hyperexponential* r.v.
    - $Pr [X <= x] = 1 – p e^{-\mu 1x} -(1-p) e^{-\mu 2x}$
    - $E[X] = p/ \mu 1 + (1–p)/ \mu 2$



Process Scheduling

- Priority queues: multiples queues, each with a different priority
  – Use strict priority scheduling
  – Example: page swapper, kernel tasks, real-time tasks, user tasks
- Multi-level feedback queue
  – Multiple queues with priority
  – Processes dynamically move from one queue to another
    - Depending on priority/CPU characteristics
  – Gives higher priority to I/O bound or interactive tasks
  – Lower priority to CPU bound tasks
  – Round robin at each level

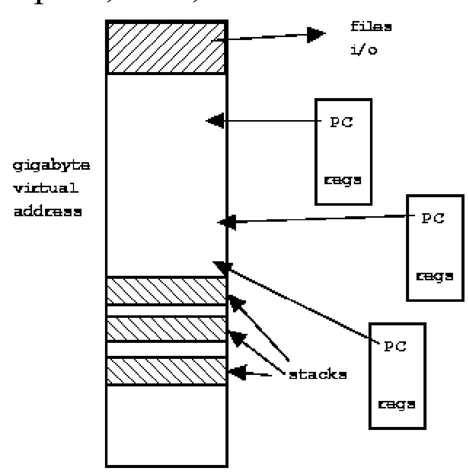


Processes and Threads

- Traditional process
  - One thread of control through a large, potentially sparse address space
  - Address space may be shared with other processes (shared mem)
  - Collection of systems resources (files, semaphores)
- Thread (light weight process)
  - A flow of control through an address space
  - Each address space can have multiple concurrent control flows
  - Each thread has access to entire address space
  - Potentially parallel execution, minimal state (low overheads)
  - May need synchronization to control access to shared variables

Threads

- Each thread has its own stack, PC, registers
  - Share address space, files,…



Why use Threads?

- Large multiprocessors need many computing entities (one per CPU)
- Switching between processes incurs high overhead
- With threads, an application can avoid per-process overheads
  - Thread creation, deletion, switching cheaper than processes
- Threads have full access to address space (easy sharing)
- Threads can execute in parallel on multiprocessors

Why Threads?

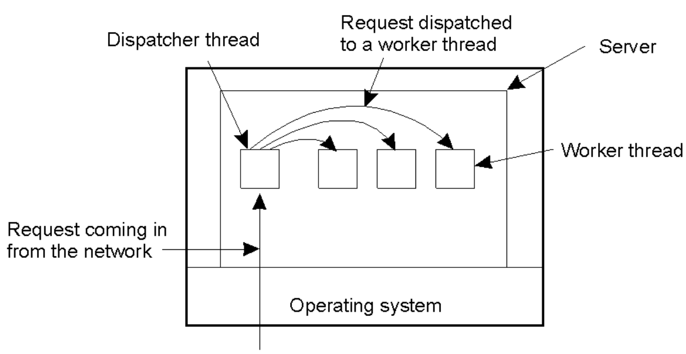
- *Single threaded process:* blocking system calls, no parallelism
- *Finite-state machine* [event-based]: non-blocking with parallelism
- *Multi-threaded process:* blocking system calls with parallelism
- Threads retain the idea of sequential processes with blocking system calls, and yet achieve parallelism
- Software engineering perspective
  - Applications are easier to structure as a collection of threads
    - Each thread performs several [mostly independent] tasks

Multi-threaded Clients Example : Web Browsers

- Browsers such as IE are multi-threaded
- Such browsers can display data before entire document is downloaded: performs multiple simultaneous tasks
  - Fetch main HTML page, activate separate threads for other parts
  - Each thread sets up a separate connection with the server
    - Uses blocking calls
  - Each part (gif image) fetched separately and in parallel
  - Advantage: connections can be setup to different sources
    - Ad server, image server, web server…

Multi-threaded Server Example

- Apache web server: pool of pre-spawned worker threads
  - Dispatcher thread waits for requests
  - For each request, choose an idle worker thread
  - Worker thread uses blocking system calls to service web request



Thread Management

- Creation and deletion of threads
  - Static versus dynamic
- Critical sections
  - Synchronization primitives: blocking, spin-lock (busy-wait)
  - Condition variables
- Global thread variables
- Kernel versus user-level threads

User-level versus kernel threads

- *Key issues:*

- Cost of thread management
  - More efficient in user space
- Ease of scheduling
- Flexibility: many parallel programming models and schedulers
- Process blocking – a potential problem

User-level Threads

- Threads managed by a threads library
  - Kernel is unaware of presence of threads
- Advantages:
  - No kernel modifications needed to support threads
  - Efficient: creation/deletion/switches don't need system calls
  - Flexibility in scheduling: library can use different scheduling algorithms, can be application dependent
- Disadvantages
  - Need to avoid blocking system calls [all threads block]
  - Threads compete for one another
  - Does not take advantage of multiprocessors [no real parallelism]

User-level threads

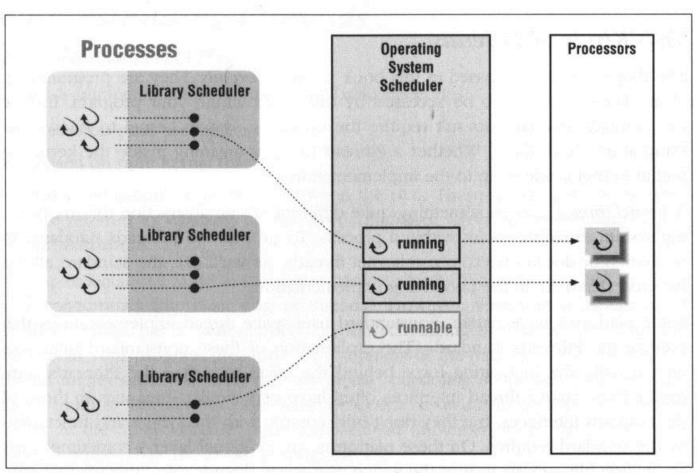


Figure 6-1: User-space thread implementations

Kernel-level threads

- Kernel aware of the presence of threads
  - Better scheduling decisions, more expensive
  - Better for multiprocessors, more overheads for uniprocessors

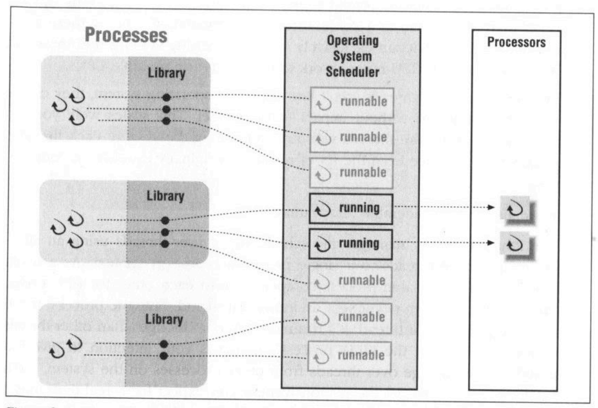


Figure 6-2: Kernel thread–based implementations

Light-weight Processes

- Several LWPs per heavy-weight process
- User-level threads package
  - Create/destroy threads and synchronization primitives
- Multithreaded applications – create multiple threads, assign threads to LWPs (one-one, many-one, many-many)
- Each LWP, when scheduled, searches for a runnable thread *[two-level scheduling]*
  - Shared thread table: no kernel support needed
- When a LWP thread block on system call, switch to kernel mode and OS context switches to another LWP

LWP Example

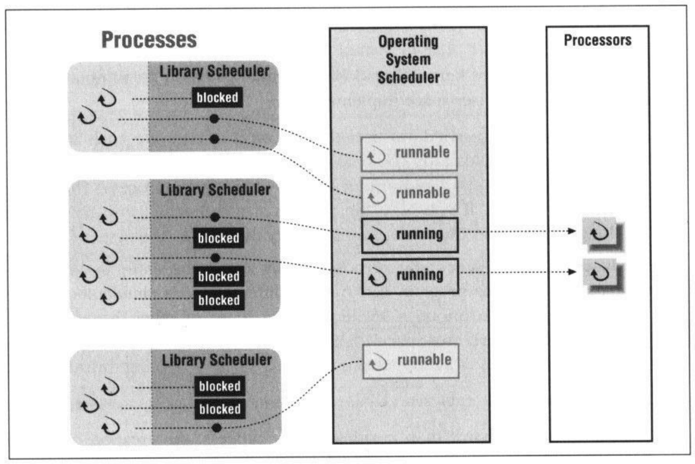


Figure 6–3: Two-level scheduler implementations

Thread Packages

- Posix Threads (pthreads)
  - Widely used threads package
  - Conforms to the Posix standard
  - Sample calls: pthread_create,…
  - Typical used in C/C++ applications
  - Can be implemented as user-level or kernel-level or via LWPs
- Java Threads
  - Native thread support built into the language
  - Threads are scheduled by the JVM