Introduction

Motivation

Reference: Vahalia 3.1

- Many applications involve several independent tasks that can be done concurrently

Examples

1. GUI based applications e.g. web browser

Motivation

Reference: Vahalia 3.1

- Many applications involve several independent tasks that can be done concurrently

Examples

1. GUI based applications e.g. web browser
2. Standard client-server setup: processing client requests concurrently

Motivation

Reference: Vahalia 3.1

- Many applications involve several independent tasks that can be done concurrently

Examples

1. GUI based applications e.g. web browser
2. Standard client-server setup: processing client requests concurrently
3. Parallelisable programs that can take advantage of multiprocessor architecture e.g. `make` utility

Motivation

Reference: Vahalia 3.1

- Many applications involve several independent tasks that can be done concurrently

  Examples

  1. GUI based applications e.g. web browser
  2. Standard client-server setup: processing client requests concurrently
  3. Parallelisable programs that can take advantage of multiprocessor architecture e.g. `make` utility

- Using multiple processes to achieve concurrency is avoidable:
  - memory load/system overhead increases substantially
  - explicit interprocess communication mechanism must be used

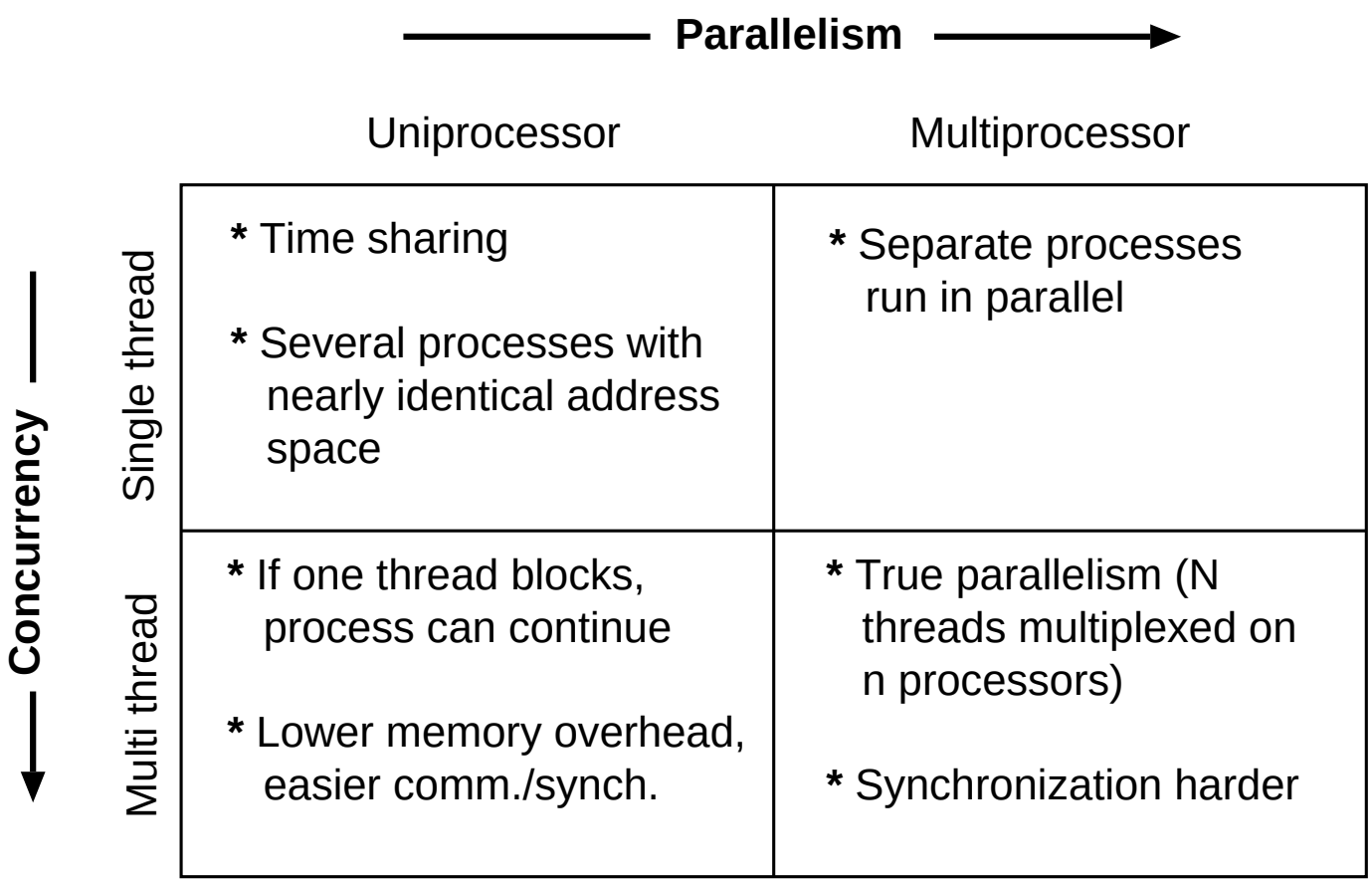
Motivation

- Solution: use lightweight processes / threads
- Thread / lightweight process $\equiv$ sub-processes within a process
- Threads : process = processes : machine
  - if a thread is blocked, another thread can run
  - timesharing + parallel execution on a multiprocessor
- Threads share the same address space
  - lightweight communication possible
  - synchronisation may be lightweight but is important
- Best for largely independent tasks that do not involve much synchronisation

Digression: parallelism vs. concurrency

- **Parallelism:** (physical)
  - actual degree of parallel execution achieved
  - limited by number of physical processors available
- **Concurrency:** (conceptual)
  - maximum parallelism achievable with unlimited processors
  - determined by application and its design

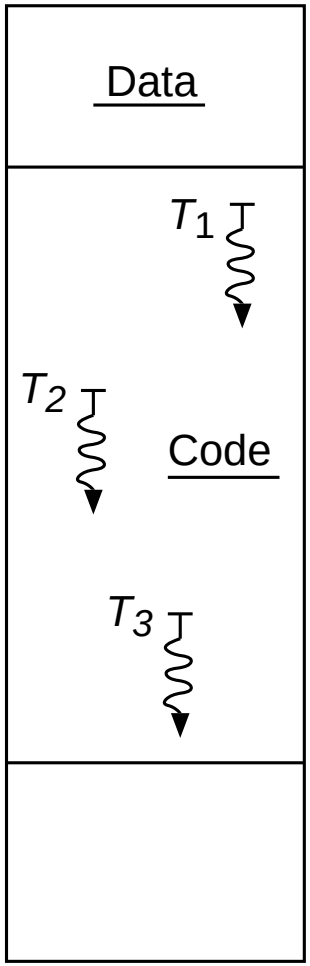
Parallelism vs. concurrency



kernel threads $\sim$ parallelism user threads $\sim$ concurrency

Basic features

- Process = set of threads + collection of shared resources
- Shared resources:
  - address space (code + data)
  - user credentials
  - open files
  - child processes
- Private resources for each thread:
  - PC, stack, register context
  - child threads
  - state
- No protection between threads $\Rightarrow$ programmer is reponsible for synchronization to prevent data corruption



User threads, kernel threads, LWPs

User threads

Reference: Vahalia 3.2.3

- Threads abstraction provided by user level library
- Library provides functions for creation, destruction, switching, scheduling of threads without kernel support
- Each user thread has:
  - user stack
  - area to save user-level register context
  - state information
  - signal mask, etc.

Can be saved and restored without kernel intervention.

User threads

Synchronisation

- Global data structures shared
  $\Rightarrow$ must be protected using synchronisation primitives e.g., lock variables/semaphore
- Thread library provides implementation of semaphores (or similar)
- Synchronisation operations can block threads and switch to other thread (if necessary)

User threads

Synchronisation

- Global data structures shared
  $\Rightarrow$ must be protected using synchronisation primitives e.g., lock variables/semaphore
- Thread library provides implementation of semaphores (or similar)
- Synchronisation operations can block threads and switch to other thread (if necessary)

Busy waiting vs. blocking

- busy waiting = user mode operation $\Rightarrow$ low overhead
- good option for small critical sections / resources that are held only briefly

User threads: asynchronous I/O

- Allows processes to perform I/O without blocking
- Read/write request simply queues the operation and returns; when I/O completes, process is informed via SIGPOLL
- Programming using AIO is complex
- Threads library uses asynchronous methods internally and provides applications a synchronous programming environment
  - each request is synchronous w.r.t. calling thread (thread blocks until I/O completes)
  - library invokes asynchronous I/O operation and schedules another thread
  - on I/O completion, library reschedules blocked thread

NOTE: difference between asynchronous I/O and non-blocking I/O

User threads: advantages

- Natural, synchronous programming model
- Thread operations / interactions involve only user-level context
  - no system call / kernel mode switch required
    $\Rightarrow$ extremely lightweight (fast, low memory overhead)
  - Example:

    SPARC 2: user thread creation - 50-60μs
    process creation - 1700μs

figures are very old!

User threads: disadvantages

- Kernel schedules processes without knowledge of constituent threads / thread-level priorities
  - when process is pre-empted, all its threads are pre-empted
  - process running a high priority user thread may be pre-empted in favour of a process running a low priority thread
  - thread holding spin lock may be pre-empted in favour of thread trying to acquire the lock
    $\Rightarrow$ scheduled thread wastes CPU time by busy-waiting
- No parallelism even on a multiprocessor
- Thread switching
  - clock interrupts occur periodically $\Rightarrow$ scheduler can be run
  - once a thread starts running, no other thread will run until the thread voluntarily gives up CPU (calls thread library function)

This situation actually only happens when user threads run on top of separate LWPs.

Kernel threads

Reference: Vahalia 3.2.1

- Created/destroyed as needed by the kernel for executing a specific function
- *Not visible to user programs*
- Shares kernel text, global data
- Private resources:
  - thread table entry
  - kernel stack
  - register context
  - scheduling / synchronization info
- Relatively inexpensive to create
- Context switching is quick (memory mappings do not have to be changed)

Kernel threads

Examples: some system processes (e.g., daemons)

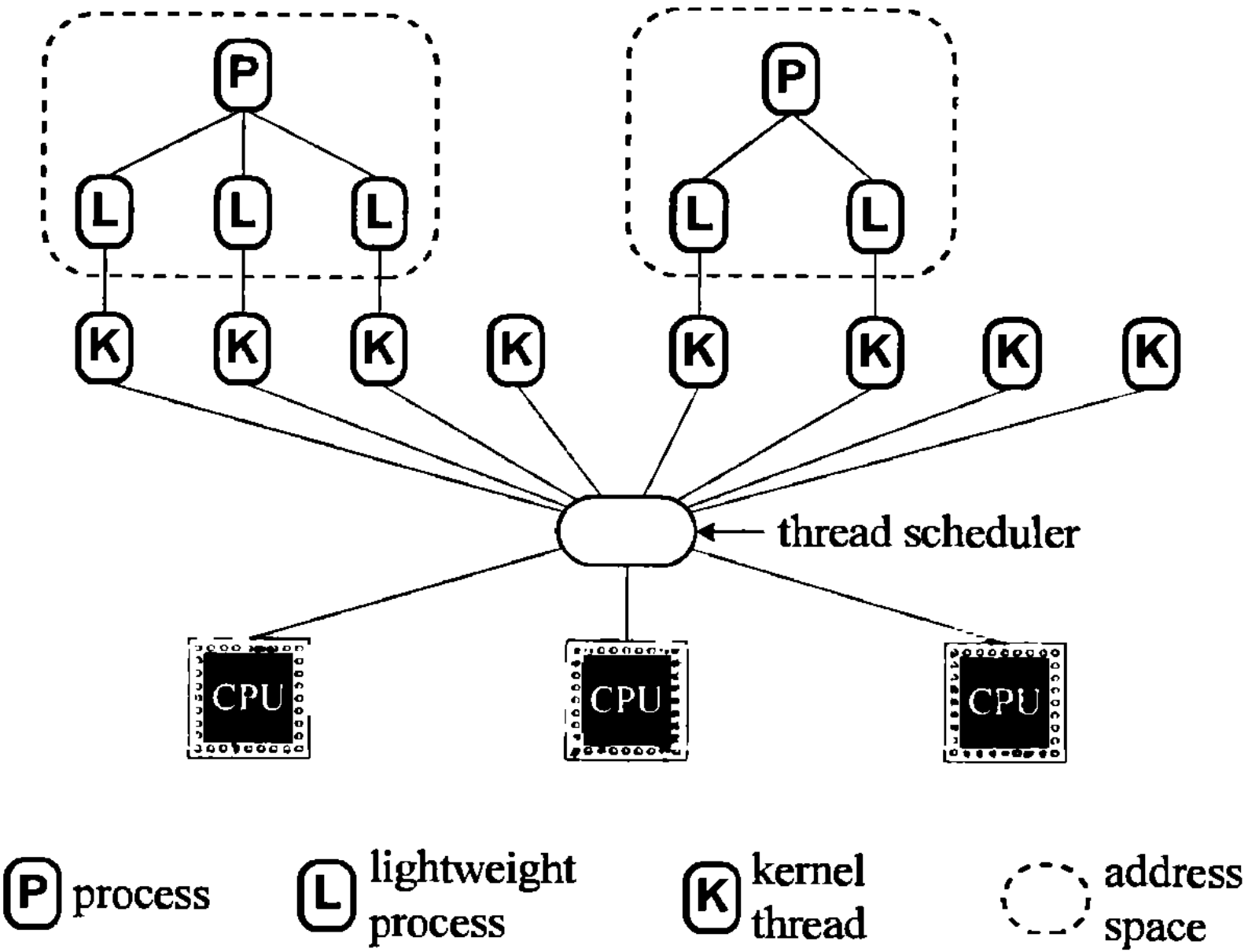
- Start at user level
- Execute (almost) entirely in kernel mode
  $\Rightarrow$ functionally equivalent to kernel threads
  (process specific admin. info is not used)
- Implemented as *processes* in systems that do not have kernel threads
- Implemented as *kernel threads* in modern multi-threaded kernels

Lightweight processes

Reference: Vahalia 3.2.2

- LWP $\equiv$ kernel-supported user thread / *kernel thread "visible" to users*
- Process contains one or more LWPs
  - each LWP is supported by a separate kernel thread
  - LWPs share address space and other resources of process

Lightweight processes



Lightweight processes

- Independently scheduled by kernel
- Can be dispatched to run on different processors on multiprocessor systems
- Resource or I/O wait blocks individual LWPs (not entire process)
- Access to shared data has to be synchronized
  - if an LWP tries to access locked data, it will block / busy-wait

LWP: disadvantages

- Creation/destruction/synchronisation/scheduling of LWPs require system calls $\Rightarrow$ higher overhead
  - mode switch + copy between user and kernel address space required
  - unsuitable for applications that
    - use a large number of threads
    - create/destroy threads frequently
    - control is frequently transferred from thread to thread
- Most useful if each thread is fairly independent of the others

  (frequent access of shared data $\Rightarrow$ synchronization overhead $\uparrow$)

Design issues

Design issues: stack growth

Reference: Vahalia 3.3.5

- Single-threaded process:
  - dedicated stack segment
  - stack overflow $\rightarrow$ protection fault $\rightarrow$ kernel automatically extends stack (instead of sending a signal)
- Multi-threaded process:
  - several user stacks
  - stacks allocated by threads library, possibly from heap/data region
  - library may protect against overflow by allocating a write-protected page just beyond the end of stack
  - stack overflow $\rightarrow$ SIGSEGV $\rightarrow$ thread handles it appropriately

Design issues: fork

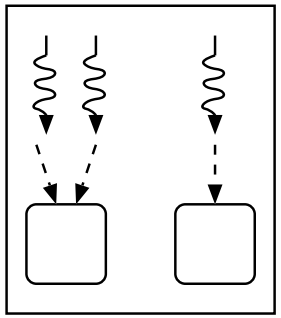
Reference: Vahalia 3.3.1

Duplicate all LWPs of parent or only the one that invokes fork?

Option1: Copy only the calling LWP into the new process

Advantages:

- more efficient
- preferable if child calls *exec* to invoke another program after *fork*



Disadvantages: LWPs may be used to support user-level thread libraries (user thread $\equiv$ data structure in user space)

- new process may contain user-level threads not bound to any LWP
- if child process tries to acquire locks held by a non-existent thread, deadlock may occur

See `http://www.alexonlinux.com/cautionary-tale-about-using-threads-and-fork` for a report of this occurring in practice.

Design issues: fork

Duplicate all LWPs of parent or only the one that invokes fork?

Option2: Copy entire process (all LWPs)

Advantages:

- preferable when the entire process needs to be cloned (rather than *exec*)

Disadvantages:

- if cloned LWP is manipulating shared data structures, then shared data may become corrupted

Design issues: fork

Duplicate all LWPs of parent or only the one that invokes fork?

Choice made by POSIX standard:

- Only the thread calling `fork()` is duplicated
- Any other threads that existed in the parent *never run* in the child
- Stacks corresponding to other *continue to exist* in the child
- Recommendation: `fork()` should be called from a thread in a multi-threaded process only if `fork()` is followed almost immediately by `exec()`
  - any code between `fork()` and `exec()` should not access resource outside the thread
    e.g., no `malloc()` calls

Design issues: fork

Duplicate all LWPs of parent or only the one that invokes fork?

Choice made by Solaris (POSIX compliant):

- Provide `forkall()` in addition to `fork()`: copies all threads into the child;
- threads continue running as before in child process
- Use cases for `forkall()`: ???

Choice made by older versions of Solaris:

- Provide `fork1()` in addition to `fork()`: only copies calling thread into the child
- `fork()` duplicates entire parent process, including all threads and lightweight processes (LWPs)

Design issues

Reference: Vahalia 3.3.4

Visibility: Should LWPs be visible outside the process?

- Not visible to other processes
- LWPs within a process can see / signal each other