Introduction
Motivation

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

Examples

1. GUI based applications e.g. web browser
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
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
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 ≡ sub-processes within a process
- Threads : process = processes : machine
  - if a thread is blocked, another thread can run
  - timesharing + parallel execution on a multiprocessor
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

<table>
<thead>
<tr>
<th></th>
<th>Uniprocessor</th>
<th>Multiprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Concurrency</strong></td>
<td><strong>Single thread</strong></td>
<td><strong>Multi thread</strong></td>
</tr>
<tr>
<td><strong>Time sharing</strong></td>
<td>* Time sharing</td>
<td>* Separate processes run in parallel</td>
</tr>
<tr>
<td><strong>Synchronization</strong></td>
<td>* Several processes with nearly identical address space</td>
<td></td>
</tr>
<tr>
<td><strong>Thread Blocking</strong></td>
<td>* If one thread blocks, process can continue</td>
<td>* True parallelism (N threads multiplexed on n processors)</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td>* Lower memory overhead, easier comm./synch.</td>
<td>* Synchronization harder</td>
</tr>
</tbody>
</table>

**Parallelism**

- Uniprocessor
- Multiprocessor
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 ⇒ programmer is responsible for synchronization to prevent data corruption
User threads, kernel threads, LWPs
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
- signal mask
- state information, etc.

Can be saved and restored without kernel intervention.
Synchronisation

- Global data structures shared
  ⇒ 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: 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
User threads: advantages

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

**Example:**
SPARC 2: user thread creation - 50-60\(\mu s\)
process creation - 1700\(\mu s\)
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

- No parallelism even on a multiprocessor

- Thread switching
  - clock interrupts occur periodically ⇒ 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)
Kernel threads

- 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: system processes

- Implemented as *processes* in traditional systems since there is no provision for kernel threads
  - daemon processes start at user level, but execute entirely in kernel mode ⇒ functionally equivalent to kernel threads
    (process specific admin. info == unnecessary overhead)

- Implemented as *kernel threads* in modern multi-threaded kernels
Lightweight processes

Reference: Vahalia 3.2.2

- LWP ≡ 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

- **P** process
- **L** lightweight process
- **K** kernel thread
- **address space**

Diagram showing the relationship between processes and threads.
Lightweight processes

- LWPs are independently scheduled by kernel
- On a multiprocessor, each LWP can be despatched to run on a different processor
- 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
  - busy waiting
    - user mode operation $\Rightarrow$ low overhead
    - good option for small critical sections / resources that are held only briefly
LWP: disadvantages

- Creation/destruction/synchronisation/scheduling of LWPs require system calls
  - 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

- LWPs are 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

- Single-threaded process:
  - dedicated stack segment
  - stack overflow → protection fault → 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 → SIGSEGV → thread handles it appropriately
Duplicate all LWPs of parent or only the one that invokes fork?

**Option 1:** 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 that are not bound to any LWP
- if child process tries to acquire locks held by a non-existent thread, deadlock may occur
Design issues: fork

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

**Option 2:** 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

Visibility: Should LWPs be visible outside the process?

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

Reference: Vahalia 3.3.4
Solaris threads
Solaris/SVR4 threads

Reference: Vahalia 3.6
Kernel threads:
- Solaris kernel is organized as a set of kernel threads
- Kernel threads independently scheduled / dispatched
- May run LWP or execute internal kernel function (i.e. not associated with any process)
- Fully pre-emptible
- Context switching between threads is less expensive than context switching between processes (virtual address space does not have to be remapped)
Kernel thread specific resources:

- stack, pointer to stack
- saved copy of kernel registers
- priority / scheduling information
- pointers to connect thread record in a scheduler queue / blocked queue
- pointers to associated lwp and proc structures
- pointers to maintain a queue of all threads in a process, all threads in the system
**Lightweight processes:**
- Bound to its own kernel thread throughout its lifetime
  - LWPs are scheduled independently and may execute in parallel on multiprocessors

- Traditional *proc* structure + *u area* replaced by:
  - *proc* structure – holds all per-process data (including process specific part of traditional *u area*)
  - *lwp* structure to hold all per-LWP data
    - saved values of user-level registers
    - system call arguments, results, error code
    - signal handling information
    - resource usage, timing information, profiling data
    - alarms
    - pointers to kernel thread structure + parent *proc* structure
User threads:
- Implemented by the threads library
  - provides commonly used API
  - threads created, destroyed, managed without kernel interference
- Run on top of LWPs
  - details taken care of by threads library
    - library creates a pool of LWPs
    - all user threads are multiplexed on this pool of LWPs
    - threads may be *bound* to a dedicated LWP, or *unbound*
- relation between LWPs and user threads similar to relation between standard I/O library routines (high-level API) and UNIX systems calls (low-level API, more control)
Solaris/SVR4 threads

- Single LWP created by kernel when program is started; executes thread compiled as the main program
- additional threads created by library calls

```c
thread_id_t thread_create(char *stack_addr,
                          u int stack_size,
                          void (*func)(), void *arg,
                          int flag)
```

where `flag` determines whether new LWP is to be created, thread is to be permanently bound to this LWP
Thread data structure:
- thread id (allows threads within a process to communicate with each other)
- saved register state
- user stack - allocated by library
- signal mask
- within process priority - used by thread scheduler (not known to kernel)
- thread local storage - statically allocated data that is not shared between threads
  #pragma unshared errno
  extern int errno;
System calls:

- **fork**
  - duplicates each LWP of parent in child
  - any LWPs that were in the middle of a system call return with EINTR error

- **fork1**
  - duplicates only the thread that invokes the function
  - useful when child process expects to invoke new program

- **pread, pwrite**
  - enables concurrent random I/O by taking seek offset as an argument

- **exec**
  - first forces all but the calling LWP to exit