Threads

Processes

- processes bring extra load to the operating system:
  - process creation
  - context saving / switching
  - determining a process to run / loading a pocess
- these actions require the operating system to be active

Processes

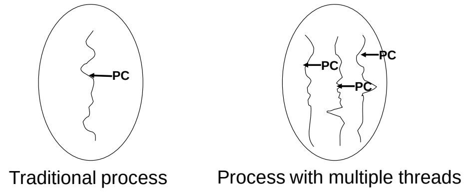
- in traditional operating systems, all processes have:
  - a private address space
  - single flow control
- in some cases more than one flow control may be required in the same address space
  - parallel processes running in the same address space

The Thread Model

- threads = light processes
- may be seen as parallel processes sharing the same address space
- makes it possible to perform more than one operation in one process

Threads

Extends the process model:

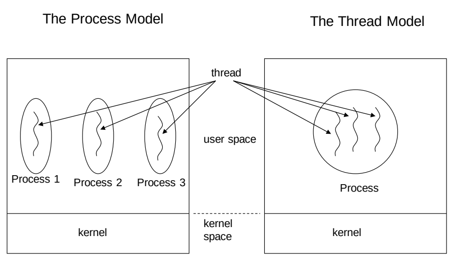


A group of threads share the same address space.

The Thread Model

- threads access and share all the resources of the process they are created in:
  - address space, memory, open files, ...
- multi threading:
  - a process may have more than one thread
  - threads are executed in order
  - context switching cost is lower
  - when a thread is blocked another one continues execution

The Thread Model



The Thread Model

- unlike processes, threads are not independent of each other:
  - they share the same address space
    - share global variables
    - access and change each other's stack
    - no protection because:
      - not possible
      - not necessary

The Thread Model

- shared by threads:
  - address space
  - global variables
  - open files
  - child processes
  - pending alarms
  - signals and signal handlers
  - accounting information
- private for each thread:
  - program counter
  - registers
  - stack
  - status

The Thread Model

- in the case of mainly independent tasks ⇒ choose the process model
- in the case of highly dependent tasks which need to be executed together ⇒ choose the thread model

The Thread Model

- thread states = process states
  - running
  - suspended (blocked)
    - waits for an external event or another thread
  - ready

Stack Usage

- each thread has a stack
- records on subroutine calls (e.g. return address) and local variables may be on stack
- threads may make different subroutine calls
  - return addresses are different ⇒ separate stacks needed

Thread Creation

- initially a process has one thread
- threads create new threads using library functions
  - e.g.: `thread_create`
    - parameter: subroutine (function) to run
- newly created thread runs in the same address space
- in some operating systems there is a parent – child hierarchy among threads
  - in most operating systems the threads are equal

Destroying Threads

- threads stop running using a library function
  - e.g.: `thread_exit`
- no timer for time sharing $\Rightarrow$ threads release the processor
  - e.g.: `thread_yield`

Interaction Between Threads

- between threads
  - synchronization
  - communication

Issues in Thread Implementation

- e.g. in the `fork` system call in UNIX systems
  - if the parent process is multi threaded, will the child process have the same threads?
    - if NO: the program may not execute correctly
    - if YES:
      - what if a thread in the parent is waiting for an input, will the thread in the child wait for an input?
      - when the input is available, will it be sent to both processes?
      - similar problem exists for open network connections

Issues in Thread Implementation

- when a thread is using a file, what if another thread closes it?
- when a process sees that additional memory is required, it makes a memory request
  - what if another thread starts executing before the request is completed and the new thread sees that additional memory is required and makes a new memory request? $\Rightarrow$ two memory requests are made
- careful design and planning are required

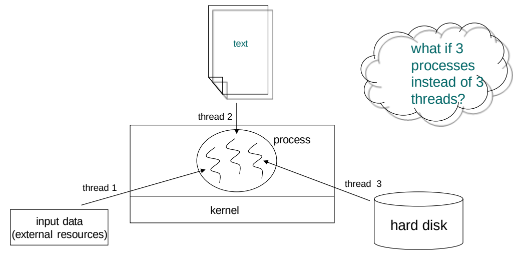
Advantages of using Threads

- a process may have more than one operation, which can be executed together
  - if some of the operations are blocked, one of the others may cotinue executing → multi threading increases performance
- thrads do not have separate resources → creating / destroying threads is easier and faster than creating / destroying processes

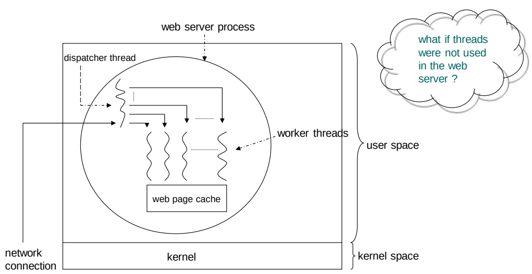
Advantages of using Threads

- if some of the threads are performing processor bound operations while some are performing I/O operations → performance increases
  - no performance improvement if all are performing processor bound operations
- suitable for multi processor systems → different threads can be assigned to different processors (parallel execution)

Example for Thread Usage – 3-Threaded Word Processor Model



Example for Thread Usage – A Web Server



Example for Thread Usage – A Web Server

Dispatcher thread code

```
while TRUE {
   get_next_request(&buf);
   handoff_work(&buf);
}
```

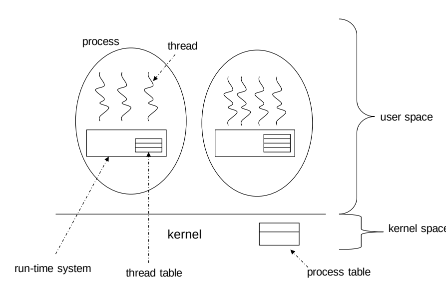
Worker thread code

```
while TRUE {
   wait_for_work(&buf);
   look_for_page_in_cache(&buf,&page);
   if (page_not_in_cache(&page))
        read_page_from_disk(&buf,&page);
   return_page(&page);
}
```

Implementing Threads

- two possible implementations
  - in user space
  - in kernel space
- hybrid implementations are also possible

Implementing Threads in User Space



Implementing Threads in User Space

- kernel is not aware of threads
- may be implemented also on operating systems which do not support multi threading
- run-time system provides a suitable execution environment:
  - thread management functions
    - `thread_create`, `thread_exit`, `thread_yield`, `thread_wait`, ...
  - thread table
    - program counter, registers, stack pointer, status information, ...

Implementing Threads in User Space

- if a thread executes an operation which causes it to be suspended (e.g. wait for another thread to finish) the thread management function performs the following:
  - change state of thread to “suspended”
  - save program counter and register contents of thread to thread table
  - take the data of the next thread from the table and load it onto the registers
  - execute the next thread

Advantages of Implementing Threads in User Space

- possible to have a separate scheduling algorithm for threads
- no space allocation required in the kernel for the thread table
- all calls are to local routines $\Rightarrow$ faster and has lower cost than making a call to a kernel routine *(system call)*

Problems with Implementing Threads in User Space

- system calls causing the thread to be suspended cause all threads to be suspended
  - the kernel suspends not only the thread but the whole process

Problems with Implementing Threads in User Space

- **solution 1:** modify system calls, but
  - modifying the operating system is not preferred
  - user progams should be modified too
- **solution 2:** in some operating systems, there is a system call which returns if a call will cause the caller process to be blocked
  - write a wrapper for system calls
  - first check if the system call will result in blocking. if yes, do not allow system call to be made, thread waits.

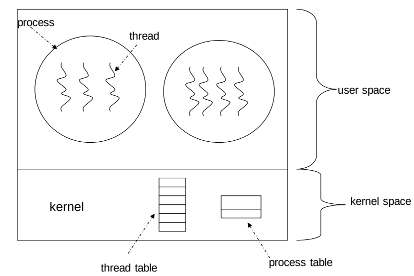
Problems with Implementing Threads in User Space

- page faults
  - if part of program code to be executed is not in memory
    - page fault occurs
    - process is suspended
    - required page is placed in memory
    - process may continue execution
  - if a thread caused the page fault
    - the kernel is not aware of threads, so the whole process is suspended

Problems with Implementing Threads in User Space

- scheduling
  - if a thread does not stop running, other threads cannot start execution
  - problems arise if both the threads and the run-time system are using timer interrupts

Implementing Threads in Kernel Space



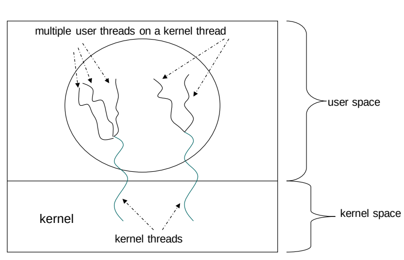
Implementing Threads in Kernel Space

- kernel is aware of threads
- thread table is kept in the kernel
- new threads are created using system calls

Implementing Threads in Kernel Space

- all calls which can cause a thread to be suspended are system calls (no need to modify system calls)
- operating system decides which thread to execute (scheduling): next thread to execute may belong to a different process
- no problems if a page fault occurs
  - the kernel executes another ready thread of the same process (if there are any)
- high cost of implementing and executing system calls
  - high execution cost if many thread creating, destroying, ... operations are performed

Hybrid Implementation of Threads



Hybrid Implementation of Threads

- kernel is only aware of kernel level threads
- more than one user thread executes on one kernel thread
- advantages and problems with user space thread implementations still exist