Thread Packages

Thomas Plagemann

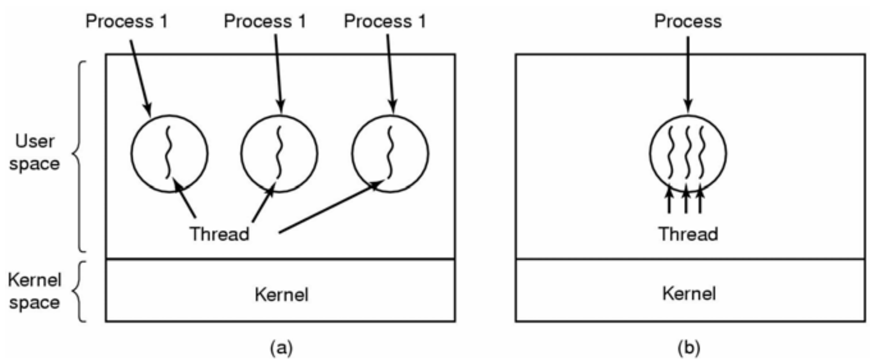
With slides from O. Anshus, C. Griwodz,
M. van Steen, and A. Tanenbaum

Overview

- What are threads?
- Why threads?
- Example: Da CaPo 1.0
- Thread implementation
  - User level
  - Kernel level
  - Scheduler activation
  - Example: Scheduler activations in NetBSD
  - Pop-up threads

Threads

The Thread Model (1)



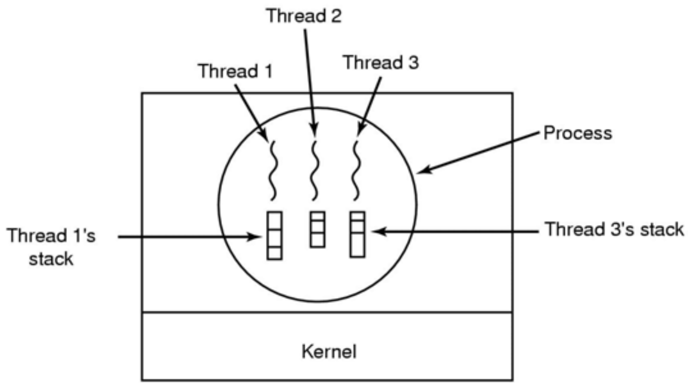
(a) Three processes each with one thread
(b) One process with three threads

The Thread Model (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

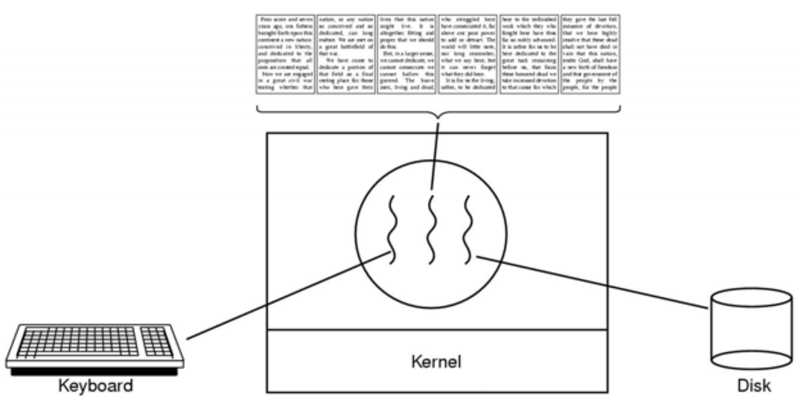
Items shared by all threads in a process	Items private to each thread

The Thread Model (3)



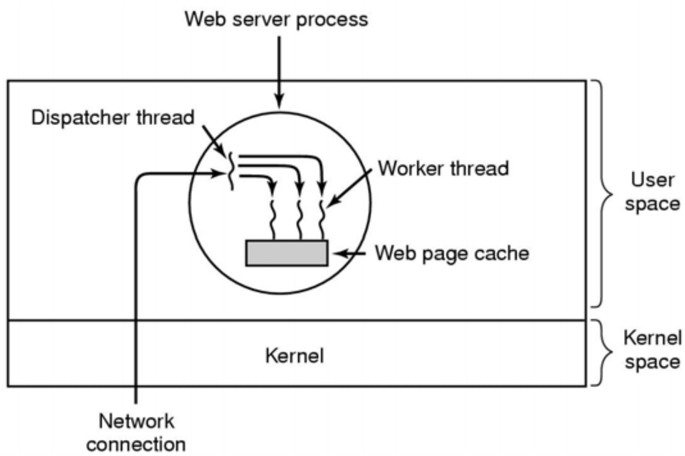
Each thread has its own stack

Thread Usage (1)



A word processor with three threads

Thread Usage (2)



A multithreaded Web server

Thread Usage (3)

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

(a)

```
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);
}
```

(b)

- Rough outline of code for previous slide
  - (a) Dispatcher thread
  - (b) Worker thread

Thread Usage (4)

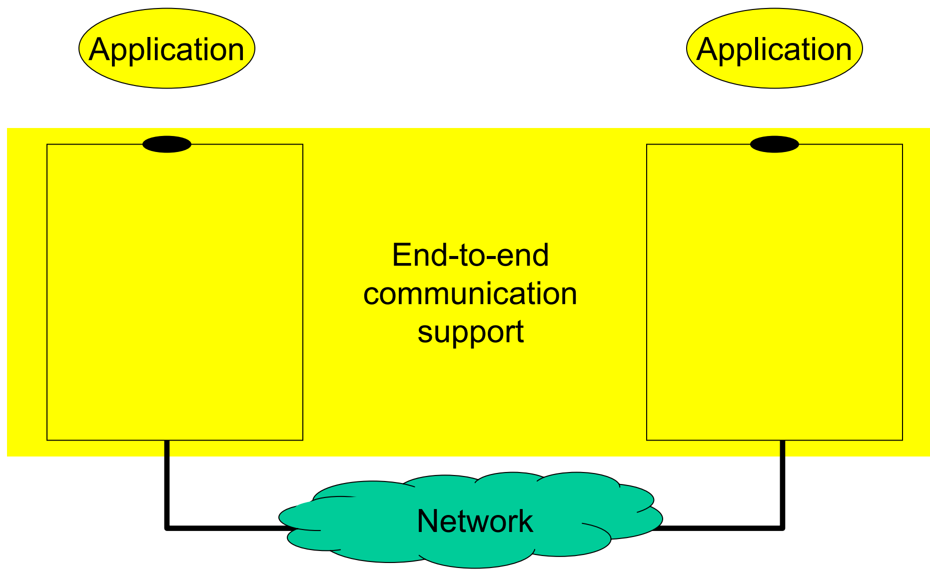
Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server

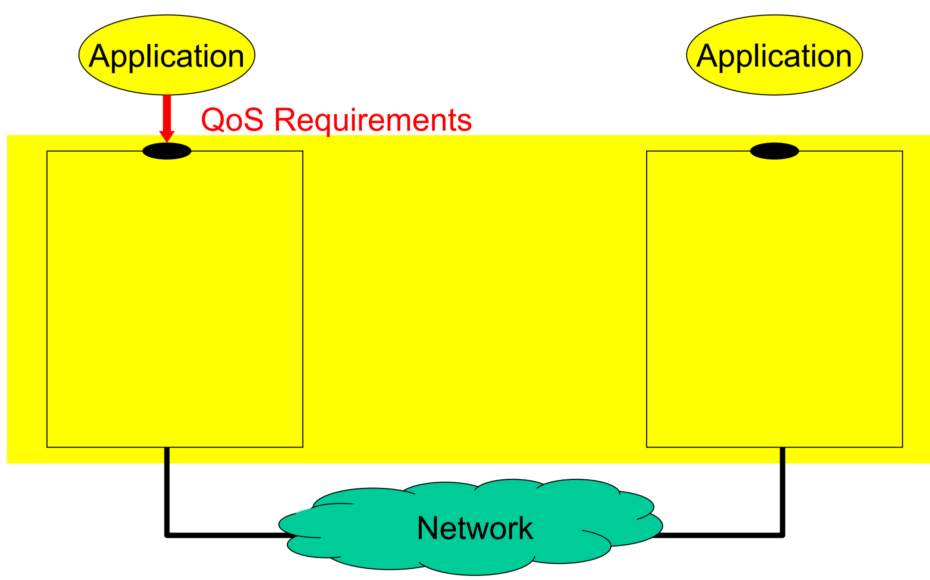
Why Threads? A lesson from Da CaPo (Dynamic Configuration of Protocols)

- General goal: flexible protocol support for multimedia applications
  - Quality-of-Service (QoS)
  - functional behaviour
- Principles:
  - decomposition of complex protocols into fine-granular micro-protocols
  - selection of optimal protocol configuration
- Related approaches: F-CSS, Adaptive, Ensemble
- Da CaPo V.1 developed for SunOS 4.1.3

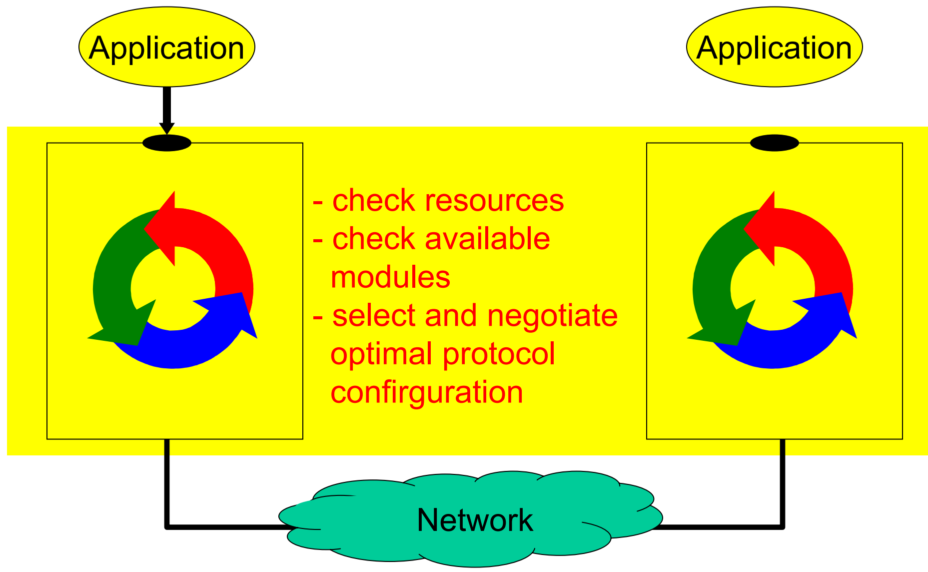
Da CaPo - Dynamic Configuration of Protocols



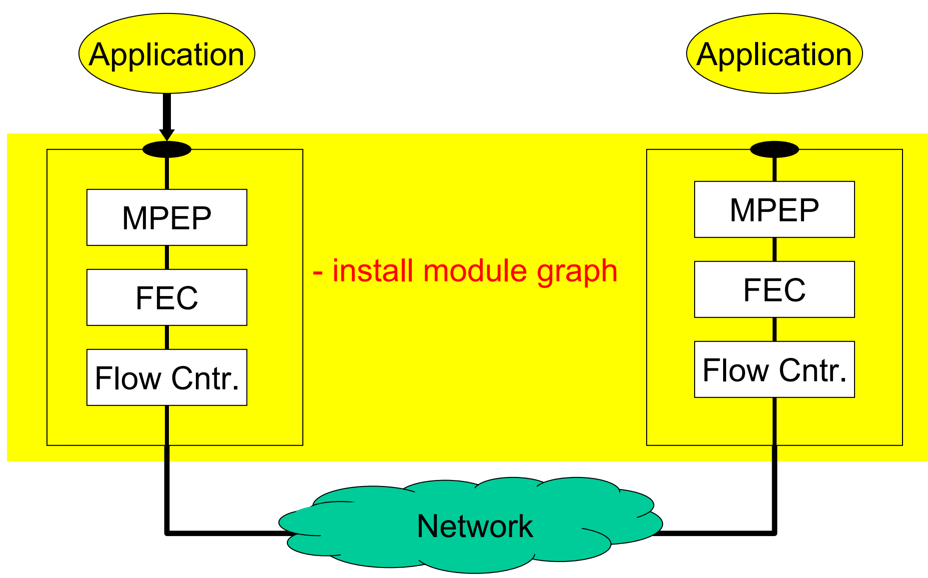
Da CaPo - Dynamic Configuration of Protocols



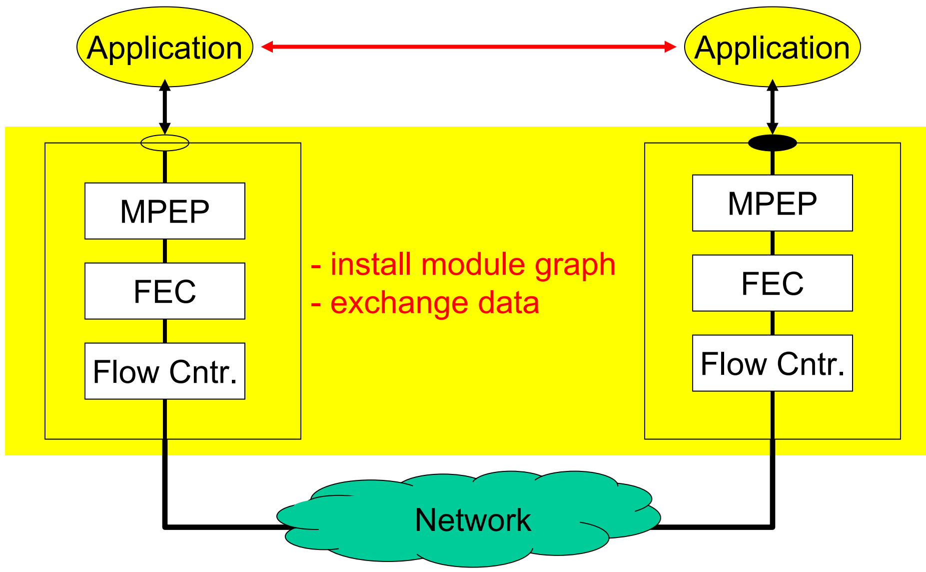
Da CaPo - Dynamic Configuration of Protocols



Da CaPo - Dynamic Configuration of Protocols

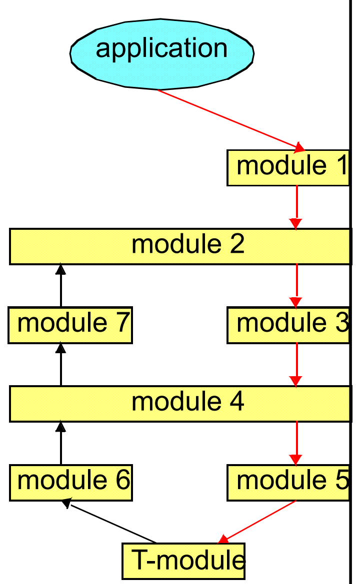


Da CaPo - Dynamic Configuration of Protocols



Da CaPo Runtime environment

- One Process per connection
- How to listen to the network and handle other modules at the same time?
- Do non-blocking reads
- Install handlers for SIGIO in T-modules
- ”Preempt” modules and ”schedule” T-module on packet arrival



Simple Example for Signal Handling

```
#include <stdio.h>
#include <signal.h>

void INThandler(int);

void main(void) {
    signal(SIGINT, INThandler);
    while (1)
        pause();
}
```

```
void INThandler(int sig) {
   char c;

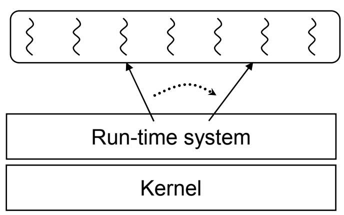
   signal(sig, SIG_IGN);
   printf("OUCH, did you hit Ctrl-C?\n“
       "Do you really want to quit? [y/n] ");
   c = getchar();
   if (c == 'y' || c == 'Y')
       exit(0);
   else
       signal(SIGINT, INThandler);
}
```

Lessons learned from this example

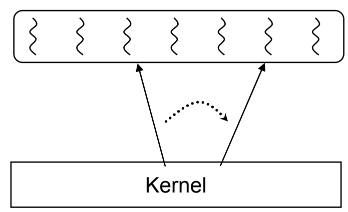
- Use of blocking system calls and threads can make programming much easier
- But you have to protect critical regions!
- Another reason: Performance!

Implementation of Thread Packages

- Two main approaches to implement threads
  - In user space
  - In kernel space



User-level thread package



Thread package managed by the kernel

Thread Package Performance

Taken from Anderson et al 1992

Operation	User level threads	Kernel-level threads	Processes
Null fork	34µs	948µs	11,300µs
Signal-wait	37µs	441µs	1,840µs

Observations

•Look at relative numbers as computers are faster in 1998 vs. 1992

•**Fork: 1:30:330**

•Time to fork off around 300 user level threads ~time to fork off one single process

•Assume a PC year 2003, '92 relative numbers = '03 actual numbers in µs

•Fork off 5000 threads/processes: 0.005s:0.15s:1,65s. OK if long running application. BUT we are now ignoring other overheads when actually running the application.

•**Signal/wait: 1:12:50**

•Assume 20M signal/wait operations: 0,3min:4 min:16,6min. **Not** OK.

Why?

•Thread vs. Process Context switching

•Cost of crossing protection boundary

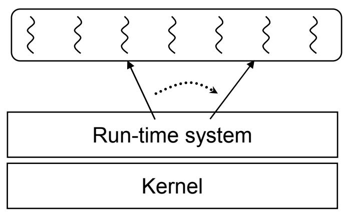
•User level threads less general, but faster

•Kernel level threads more general, but slower

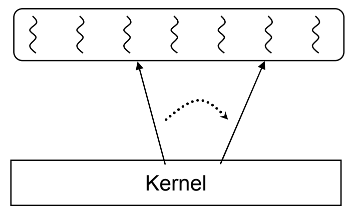
•Can combine: Let the kernel cooperate with the user level package

Implementation of Thread Packages

- Two main approaches to implement threads
  - In user space
  - In kernel space
- Hybrid solutions: cooperation between user level and kernel
  - Scheduler activation
  - Pop-up threads

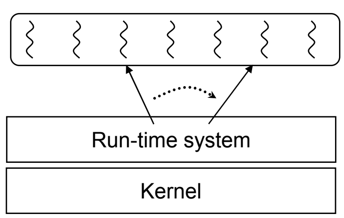


User-level thread package

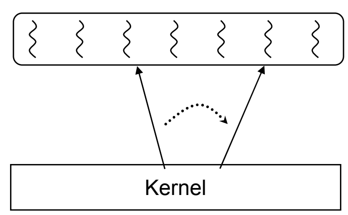


Thread package managed by the kernel

Implementation of Threads



User-level thread package



Thread package managed by the kernel

User level

•If a thread blocks in a system call, user process blocks

•Can have a wrapper around syscalls preventing process block

Kernel level

•Support for one single CPU

User level

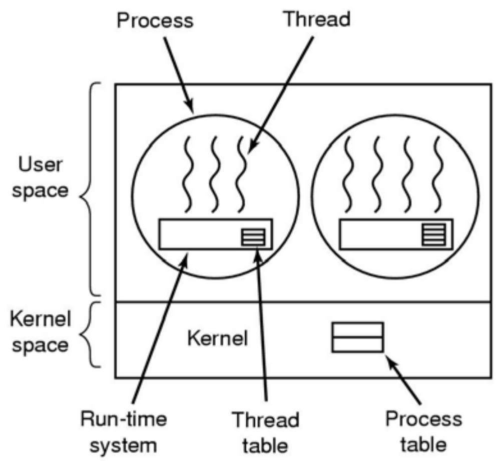
•If a thread blocks in a system call, user process does not

•Can schedule threads independently

Kernel level

•Support for multiple CPUs

Implementing Threads in User Space



A user-level thread package

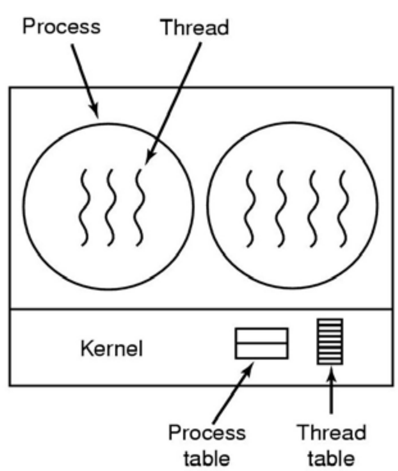
User Level Thread Packages

- Implementing threads in user space
  - Kernel knows nothing about them, it is managing single-threaded applications
  - Threads are switched by runtime system, which is much faster than trapping the kernel
  - Each process can use its own customized scheduling algorithm
  - Blocking system calls in one thread block all threads of the process (either prohibit blocking calls or write jackets around library calls)
  - A page fault in one thread will block all threads of the process
  - No clock interrupts can force a thread to give up CPU, spin locks cannot be used
  - Designed for applications where threads make frequently system calls

User Level Thread Packages

- Implementation options
  - Libraries
    - Basic system libraries (“invisible”)
    - Additional system libraries
    - Additional user libraries
  - Language feature
    - Java (1.0 – 1.2 with “green threads”)
    - ADA
    - …

Implementing Threads in the Kernel

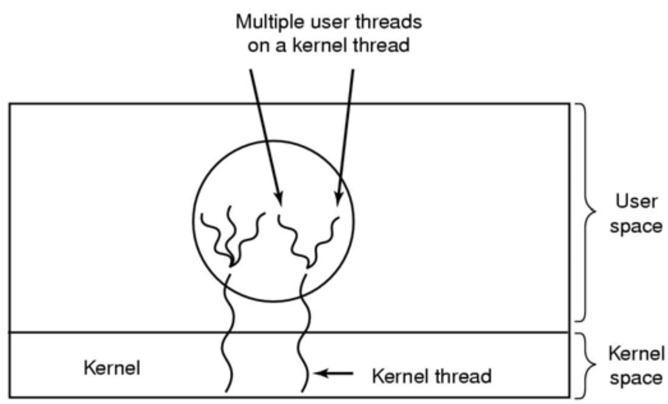


A threads package managed by the kernel

Kernel Level Thread Packages

- Implementing threads in the kernel
  - When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction (optimization by recycling threads)
  - Kernel holds one table per process with one entry per thread
  - Kernel does scheduling, clock interrupts available, blocking calls and page faults no problem
  - Performance of thread management in kernel lower

Hybrid Implementations

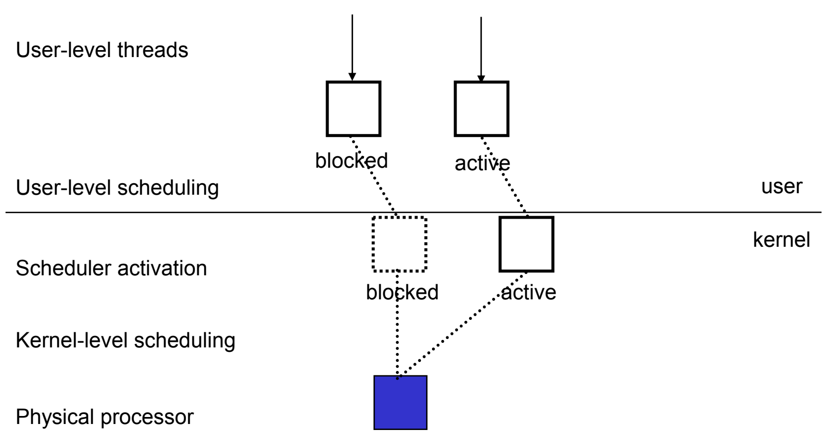


Multiplexing user-level threads onto kernel- level threads

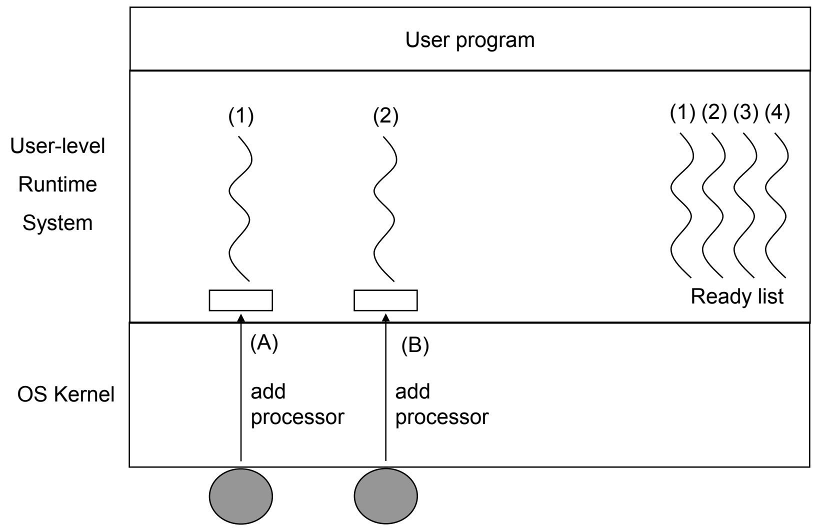
Scheduler Activations

- Scheduler activation
  - Goals: combine advantages of kernel space implementation with performance of user space implementations
  - Avoid unnecessary transitions between user and kernel space, e.g., to handle local semaphore
  - Kernel assigns virtual processors to each process and runtime system allocates threads to processors
  - The kernel informs the process's runtime system via an upcall when one of its blocked threads becomes runnable again
  - Runtime system can schedule
  - Runtime system has to keep track when threads are in or are not in critical regions
  - Upcalls violate the layering principle

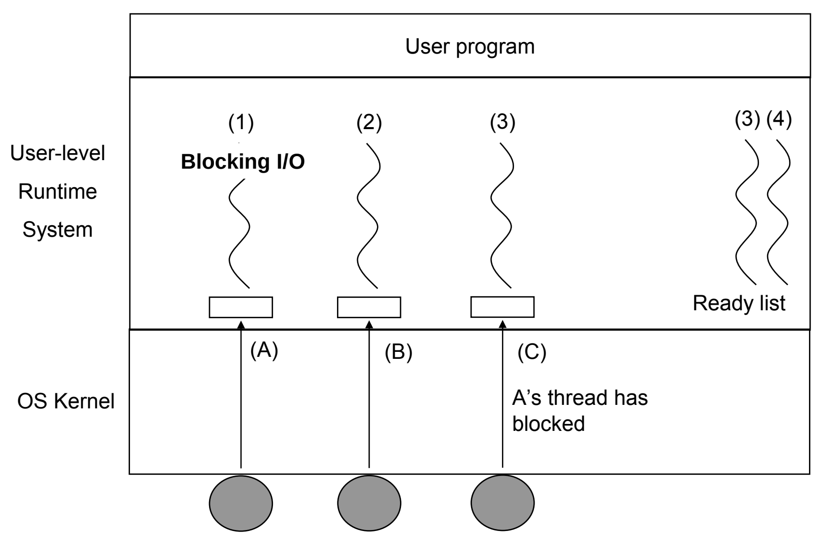
User-level threads on top of Scheduler Activations



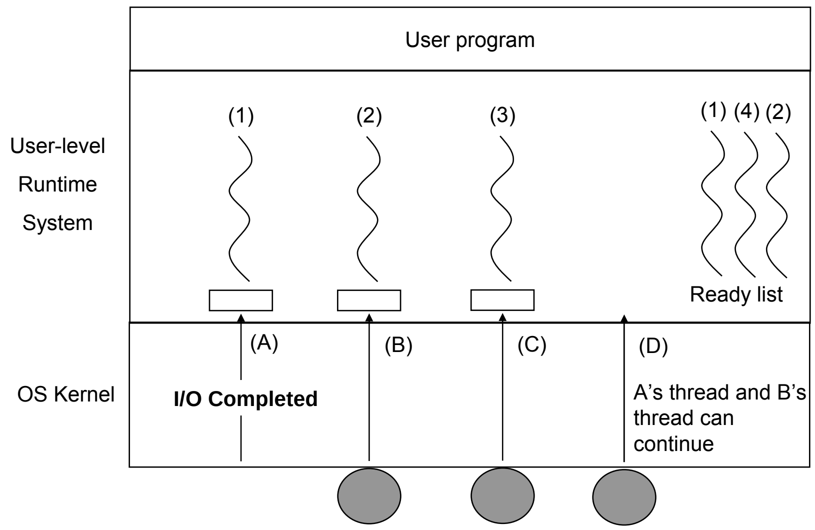
Scheduler Activations - I



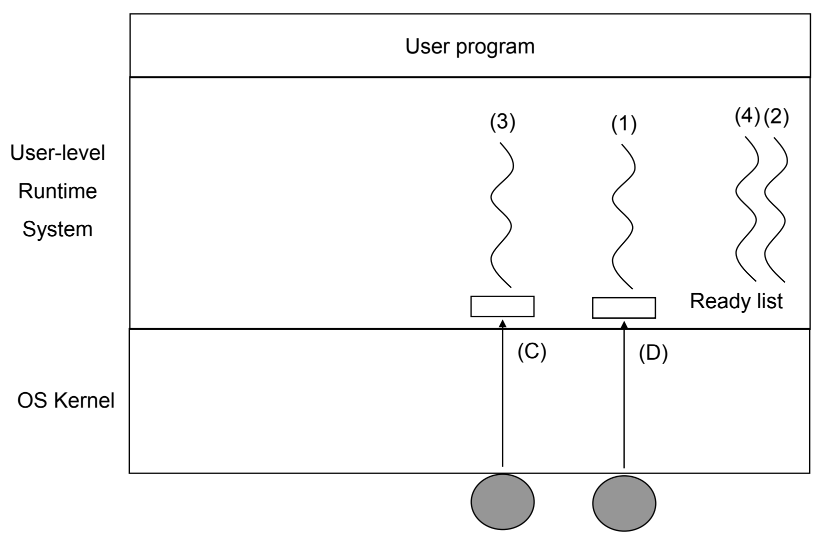
Scheduler Activations - II



Scheduler Activations - III



Scheduler Activations - IV



Scheduler Activations in NetBSD

- Nathan J. Williams: "An Implementation of Scheduler Activations on the NetBSD Operating Systems", in Proceedings of Freenix/Usenix 2002
- CVS branch `nathanw_sa`; integration into NetBSD-current in 2003
- Earlier implementations of scheduler activations in Taos, Mach 3.0, BSD/OS, Digital Unix (now Compaq Tru64 Unix)

Kernel Interface - I

- Application → scheduler activation system, i.e., by system calls:
  - `sa_register( )`
  - `sa_setconcurrency( )`
  - `sa_enable( )`
  - `sa_yield ( )`
  - `sa_prevent( )`
- Scheduler activation → application, i.e., by upcall:
  - ```
    void sa_upcall(int type,
                   struct sa_t *sas[],
                   int events,
                   int interrupted,
                   void *arg);
    ```
    ```

Kernel Interface - II

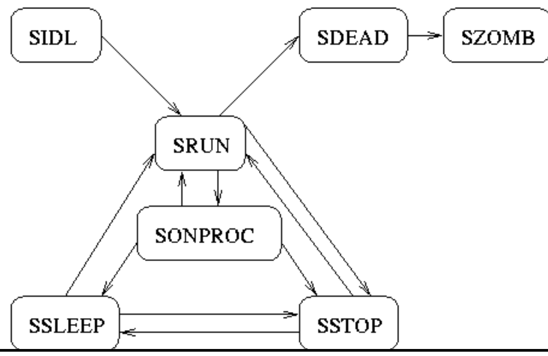
- Events that generated upcalls:
  - `SA_UPCALL_NEWPROC`
  - `SA_UPCALL_PREEMPTED`
  - `SA_UPCALL_BLOCKED`
  - `SA_UPCALL_UNBLOCKED`
  - `SA_UPCALL_SIGNAL`
  - `SA_UPCALL_USER`
- Low level upcall mechanism is similar to signal delivery

Kernel Interface - III

- Stacks:
  - Any upcall code needs to store local variables, return address, etc.
  - Using stack of preempted thread?
    - New processor allocations
    - Makes thread management more difficult
  - Each upcall got its own stack
  - System call `sa_stack( )`
- Signals:
  - Support the POSIX signal model
  - Kernel does not know about specific threads
  - Signals are handed to the application with an upcall

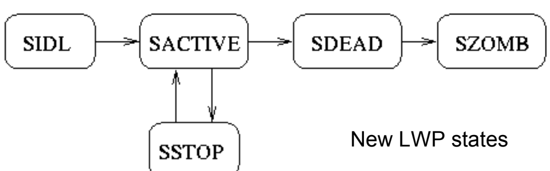
Kernel Implementation - I

- NetBSD kernel has a monolithic process structure including execution context
- First task: separation of process context from execution context
  - Old NetBSD process states

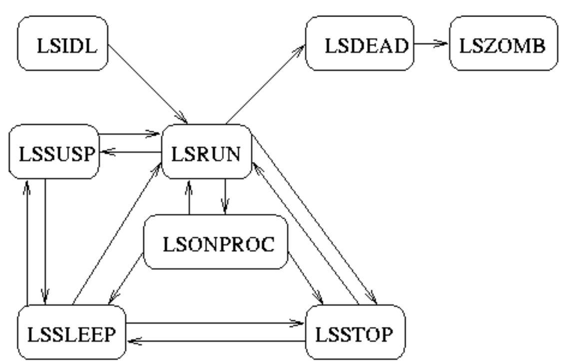


Kernel Implementation - II

New process states



New LWP states



Kernel Implementation - III

- NetBSD manages all process data in `struct proc`
- Move all execution related data into a new `struct lwp`
- Update all code parts with variables of type `struct proc`
- Scheduler must handle LWPs, `fork( )`

Thread Implementation

- Goal: become the supported POSIX compatible library for NetBSD
- Scheduler activations can always be preempted
  - Violation of atomicity of critical sections of code
  - Example maintenance of ru queue in thread library
    - Normally spin lock are used
    - Upcall handler might try to get the same lock
    - Deadlock (we discuss this later in more detail!)

Some Performance Numbers

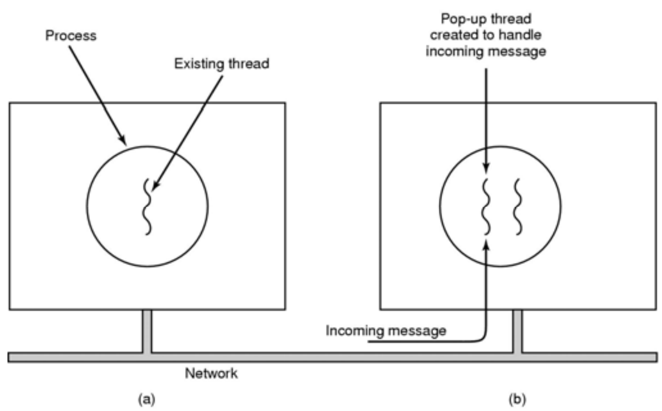
- HBench-OS on
  500 MHz Digital Alpha
  21164 system

	before SA	after SA
getpid	0.631	0.601
getrusage	4.053	4.332
timeofday	1.627	1.911
sbrk	0.722	0.687
sigaction	1.345	1.315

- Apple iBook
  500 MHz G3 CPU
  256 L2 cache

	SA	PTH	Linux
Thread	15 µs	96 µs	90 µs
Mutex	0.4 µs	0.3 µs	0.6 µs
Context	225 µs	166 µs	82 µs

Pop-Up Threads



- Creation of a new thread when message arrives
  (a) before message arrives
  (b) after message arrives

Pop-Up Threads

- Fast reacting to external events possible
  - Packet processing is meant to last a short time
  - Packets may arrive frequently

- Questions with pop-up threads
  - How to guarantee processing order without loosing efficiency?
  - How to manage time slices? (process accounting)
  - How do schedule these threads efficiently?

Existing Thread Packages

- All have
  - Thread creation and destruction
  - Switching between threads
- All specify mutual exclusion mechanisms
  - Semaphores, mutexes, condition variables, monitors

- Why do they belong together?

Some existing thread packages

- POSIX Pthreads (IEEE 1003.1c) for all/most platforms
  - Some implementations may be user level, kernel level or hybrid
- GNU PTH
- Linux
- JAVA for all platforms
  - User level, but can use OS time slicing
- Win32 for Win95/98 and NT
  - kernel level thread package
- OS/2
  - kernel level

- Basic idea in most packages
  - Simplicity, fancy functions can be built using simpler ones

Summary

- Today:
  - What are threads?
  - Why threads?
  - Example: Da CaPo 1.0
  - Thread implementation
    - User level
    - Kernel level
    - Scheduler activation
    - Example: Scheduler activations in NetBSD
    - Pop-up threads
- Tomorrow: CPU Scheduling