Thread Packages

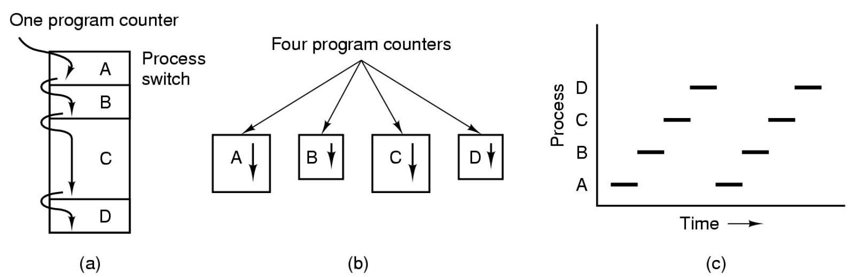
Carsten Griwodz
University of Oslo
(includes slides from O. Anshus, T. Plagemann,
M. van Steen and A. Tanenbaum)

Overview

- What are threads?
- Why threads?
- Thread implementation
  - User level
  - Kernel level
  - Scheduler activation
- Some examples
  - Posix
  - Linux
  - Java
  - Windows
- Summary

Processes

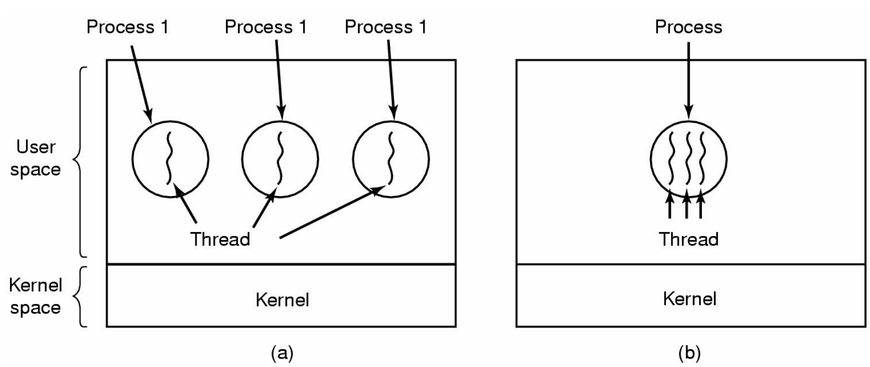
The Process Model



- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Threads

The Thread Model (1)



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

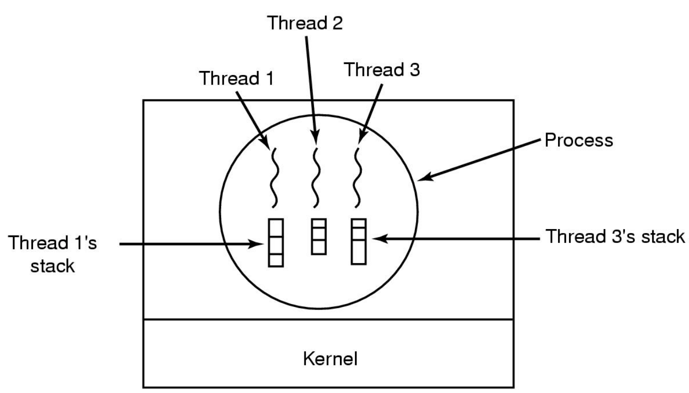
The Thread Model (2)

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

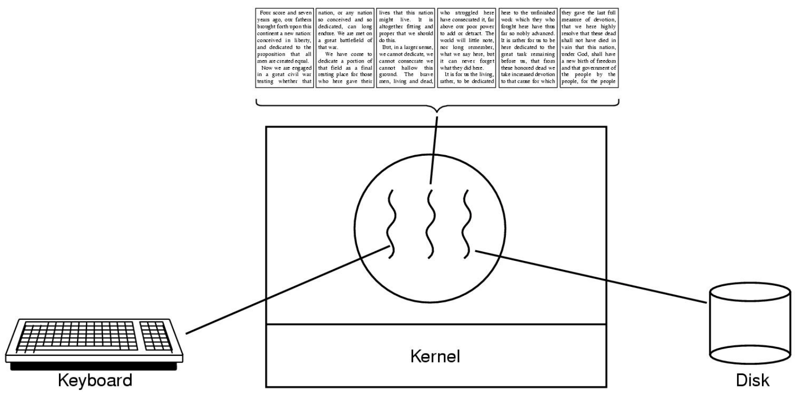
Items private to each thread

The Thread Model (3)



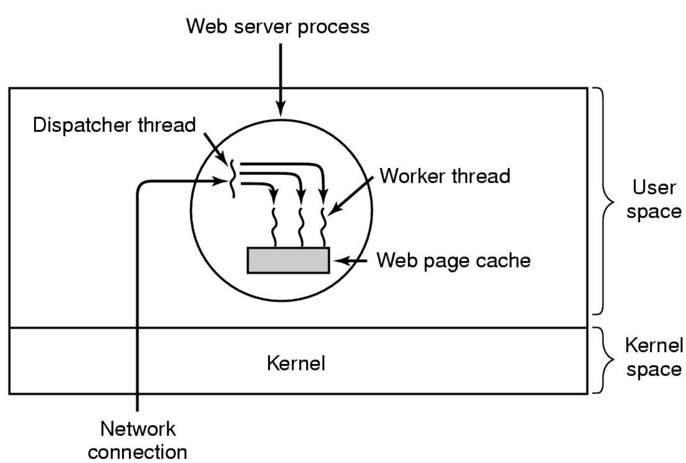
Each thread has its own stack

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

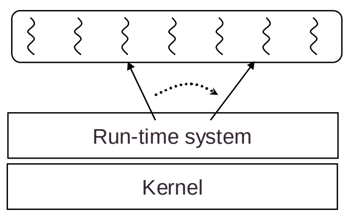
Thread Usage (4)

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

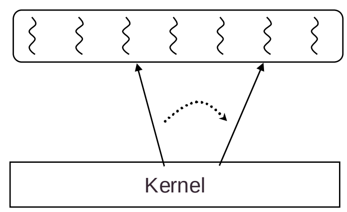
Three ways to construct a server

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

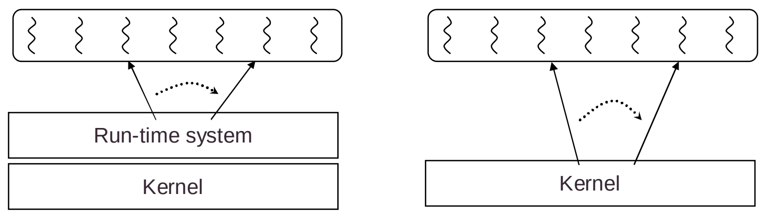
•User level threads less general, but faster

•Kernel level threads more general, but slower

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

Implementation of Thread Packages

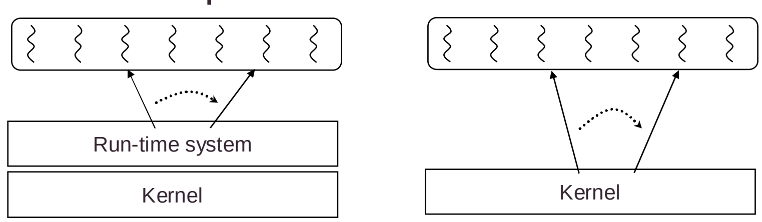
- Two main approaches to implement threads
  - In user space
  - In kernel space
- Hybrid solutions: cooperation between user level and kernel
  - Scheduler activation
  - Pop-up threads



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

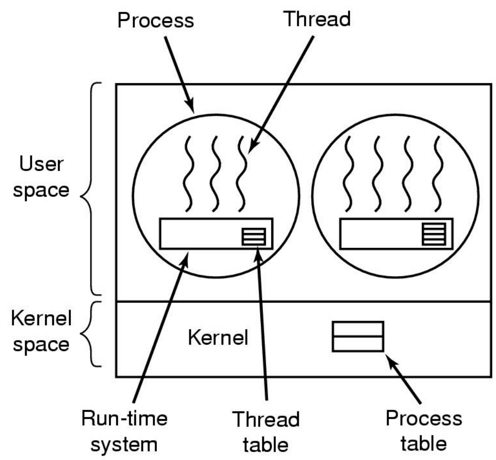
•If a thread blocks in a system call, user process does not

•Can schedule threads independently

Kernel level

•Support for multiple CPUs

Implementing Threads in User Space



A user-level thread package

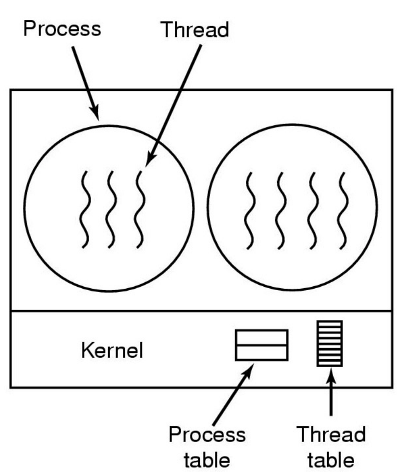
User Level Thread Packages

- 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

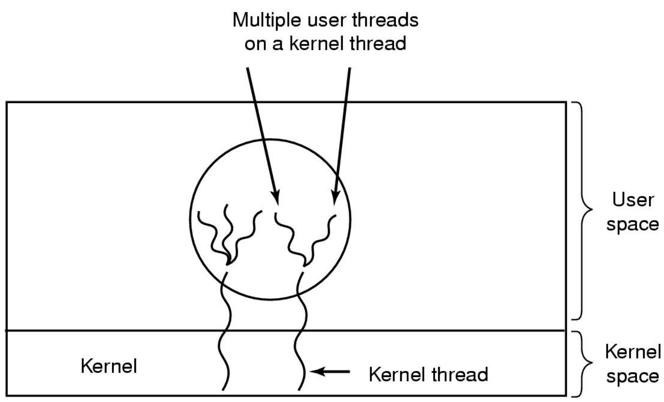


A threads package managed by the kernel

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

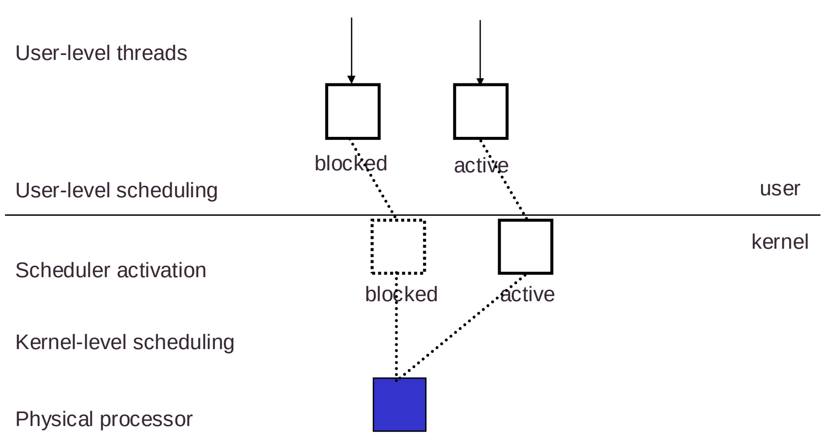


Multiplexing user-level threads onto kernel- level threads

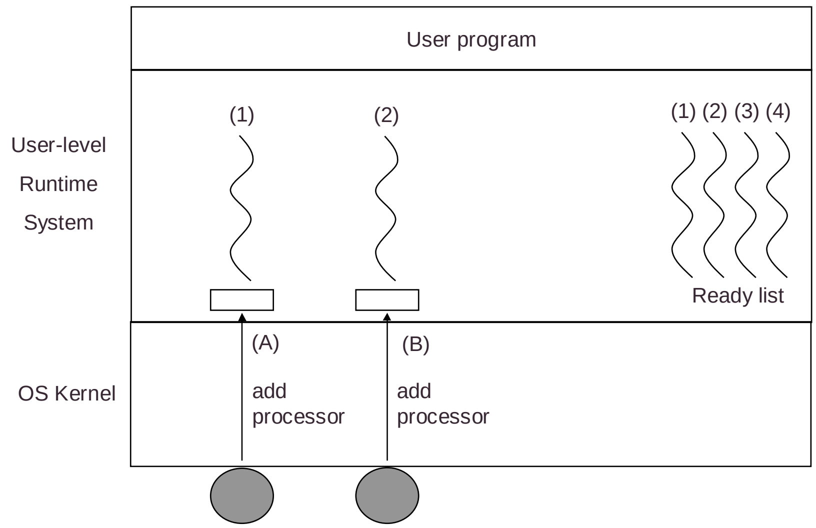
Scheduler Activations

- 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

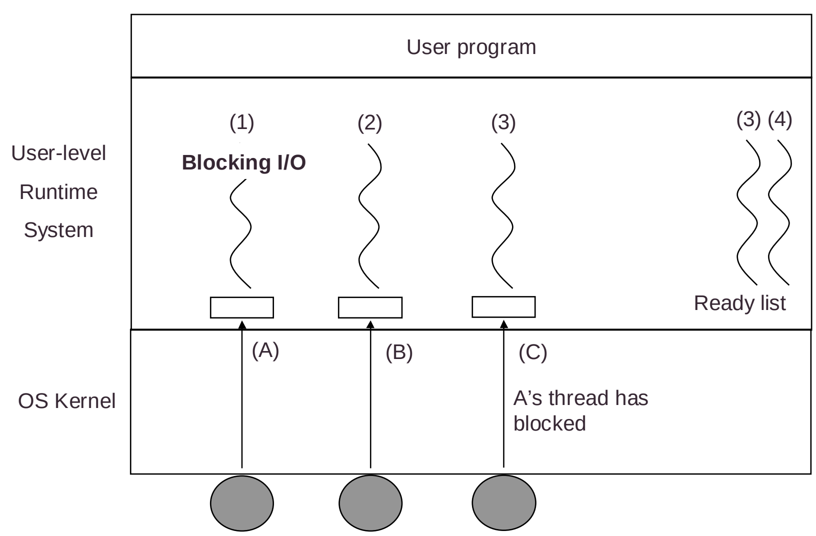
User-level threads on top of Scheduler Activations



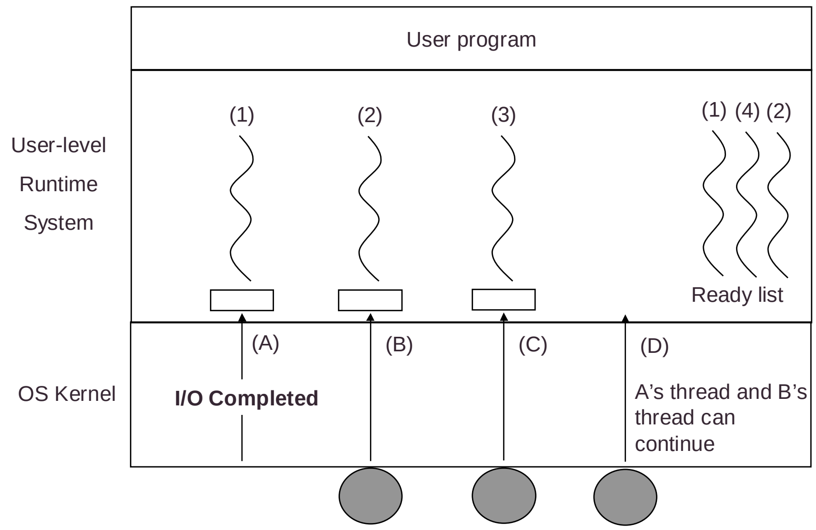
Scheduler Activations - I



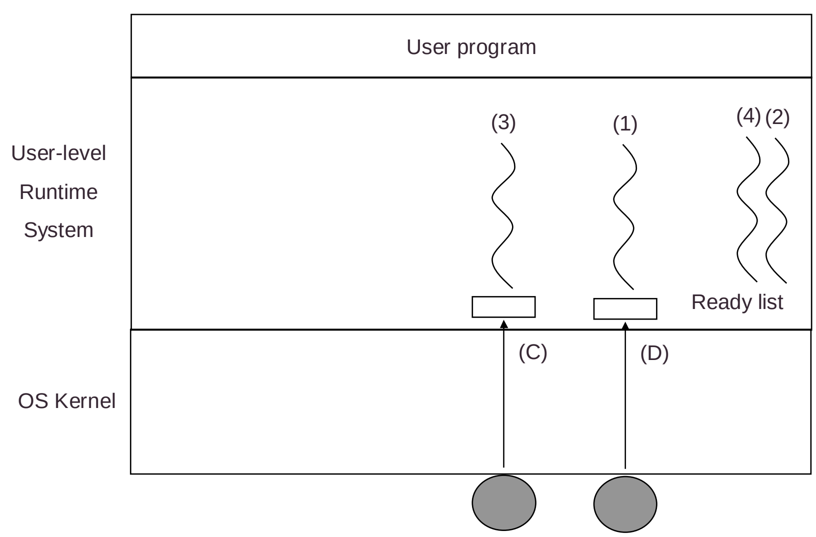
Scheduler Activations - II



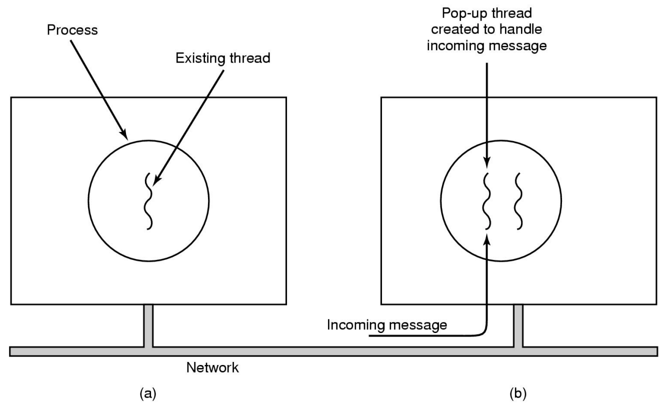
Scheduler Activations - III



Scheduler Activations - IV



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?

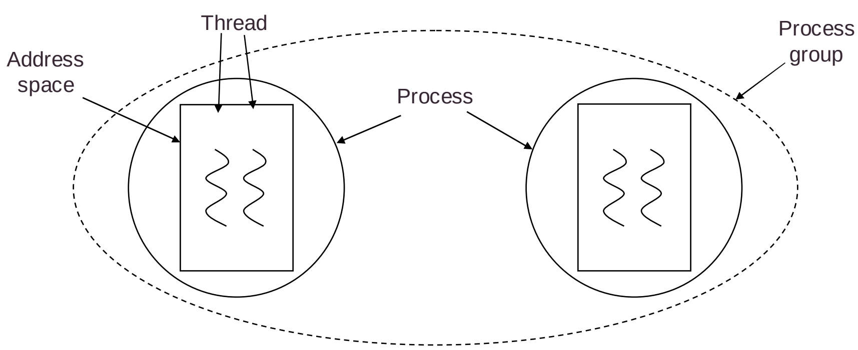
Some existing thread packages

- 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

Threads in POSIX

Thread call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release on thread waiting on a condition variable

Threads in POSIX



- Process groups: addition to simplify process management
  - Stopping process together
  - More generally signalling all processes together
  - No resource management implications

GNU PTH

- Name: Portable Threads

- User level thread package
- Implements a POSIX thread package for operating systems that don't have any
- Extends the API of the POSIX thread package
  - Many blocking functions are not wrapped by the POSIX API

GNU PTH

Thread call	Description
pth_spawn	Create a new thread
pth_wait	Wait for a generic PTH event
pth_nap	Sleep for a short time
pth_mutex_init	Create a mutex
pth_cond_init	Create a condition variable
pth_barrier_init	Create a barrier
pth_read	PTH wrapper to blocking read call
pth_select	PTH wrapper to blocking select call
pth_select_ev	Wrapper to blocking select call that can wait for other events as well, in particular mutexes etc.
…	…

Thread Package LinuxThreads

- Linux implementation is based on ideas from 4.4BSD
- New system call
- `Pid = clone(function, stack_ptr, sharing_flags, arg);`
- New thread starts executing at `function` with `arg` as parameter and a private stack
- Special feature of `clone`: **`sharing_flags`**
  - Bitmap of five bits
  - Allows much finer grain of sharing than trad. UNIX

Thread Package LinuxThreads

Flag	Meaning when set	Meaning when cleared
`CLONE_VM`	Create a new thread	Create a new process
`CLONE_FS`	Share umask, root and working dirs	Do not share them
`CLONE_FILES`	Share file descriptors	Copy the file descriptors
`CLONE_SIGHAND`	Share the signal handler table	Copy the table
`CLONE_PID`	New thread gets old PID	New thread gets own PID

Thread Package LinuxThreads

- LinuxThreads builds on `clone`
  - Processes
  - Threads
- Not POSIX compliant
  - Uses a manager thread if more than one thread exists in a process
  - LinuxThreads threads a not peers but parents ad children
  - Can not direct signals correctly at threads
  - Mutual exclusion implemented using signals

Linux NPTL

- Native POSIX Thread Library

- New thread package for Linux 2.6
- POSIX compliant

- Kernel thread implementation
  - Favored over scheduler activation approach
    - NGPT (Next Generation POSIX Threading)
  - Less code to maintain
  - Particular implementation proved to be faster

Linux NPTL

- Extends `clone`
- New mutual exclusion mechanisms
  - Rely on “fast user-level locking”
  - Wait queues are maintained by the kernel
  - Switching from kernel mode to user mode for
    - Waiting
    - Signaling if blocked processes exist

JAVA

- Multithreaded language, many packages with classes
- All threads are inside a process
- java.lang package
  - Thread class
    - start, (stop,) set priority, etc
    - synchronized keyword
- I/O in Java
  - Must create one thread per I/O channel up to Java 1.3
  - Thread will block on I/O
- Interpreted
  - (10-20 times slower than C (++))
  - … + just in time compiling at run time (closer to C(++))
  - … + portions of application can be written in C(++)

Monitors in Java

```
Public synchronized void put (int m)
{
    while (count == n)
    {
        try { wait(); }
        catch (InterruptedException e) {}
    }
    <update buffer and state variables>
    notifyAll();
}

Public synchronized void get (int m)
{
    <etc>
}
```

Monitor MUTEX

Reevaluates because all threads waiting are awaken

More on Java synchronize()

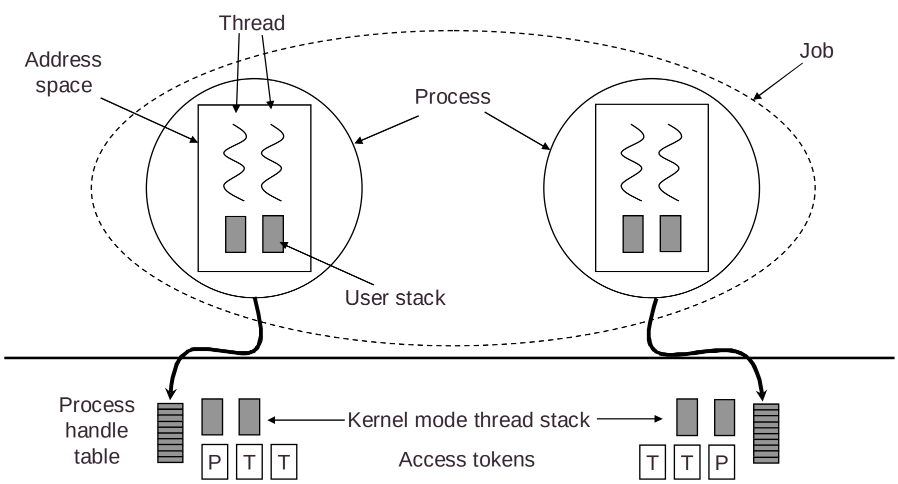
- To a block of statements (as we did in the example)
- To a method
  - Static method (a.k.a. class method)
    - Mutex on a whole class
    - Only one static synchronized method for a particular class can be running at any given time
    - Gives the thread
  - Nonstatic method
    - Mutex between different methods accessing the **same** object
    - No mutex if threads are using the same method on **different** objects

Processes and Threads in Windows 2000

Name	Description
Job	Collection of processes that share quotas and limits
Process	Container for holding resources
Thread	Entity scheduled by the kernel
Fiber	Lightweight thread managed entirely in user space

- Basic concepts used for CPU and resource management

Processes and Threads in Windows 2000



- Relationship between jobs, processes, threads and fibers

Processes and Threads in Windows 2000

Win32 API function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semaphore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitforMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled, then to non-signaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section

Summary

- What are threads?
- Why threads?
- Thread implementation
  - User level
  - Kernel level
  - Scheduler activation
- Some examples
  - Posix
  - Linux
  - Java
  - Windows
- Summary

Appendix – Java and Pthreads

- The following transparencies give more details about threads in Java and POSIX

java.lang.Thread

- **run**() is the body of the thread
- **start**()starts a thread
- **stop**() stops a thread
- **suspend**() temporarily blocks a thread
- **resume**() will resume a thread
- **sleep**() puts a thread to sleep for a specified amount of time
- **yield**() makes the current thread give up control to any other thread *of equal priority* that are waiting to run
- **join**() waits for a thread to die
- **interrupt**() wakes up a waiting thread or sets a flag on a non-waiting thread
- **interrupted**() allows a thread to test its own interrupt flag
- **isInterrupted**() allows a thread to test another threads interrupt flag
- **wait**(object) makes current thread block until **notify**(object) is called by another thread

Java: Preemptive, but not always time sliced

- A running thread will be preempted by a higher priority thread
- No guarantee that we have time slicing
  - Java assumes the OS may or may not support it for user level threads

Java Thread Groups

- A group of
  - threads
  - group of threads
- Can kill, suspend and resume ALL threads in a group with a single invocation
- Can count number of active threads
- Examples
  - Kill all threads pulling in data for a page (we clicked stop on the browser)
  - A computation is finished, so must kill all threads still computing along various branches

ThreadGroup g=new ThreadGroup(parent, name)

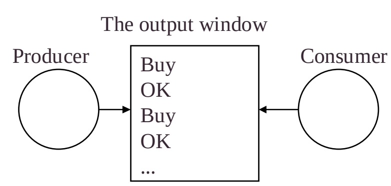
g.stop()

Int activeCount()

Types of use of Java Threads

- Unrelated threads — Unrelated, no interaction
- Related but unsynchronized threads — Work is split, but no direct interaction
- Mutually exclusive threads — Mutex
- Communicating mutually exclusive — Mutex and Condition synchronization

Unrelated & Related Unsynchronized Java Threads




```
Public class ProducerConsumer {
   public static void main (...) {
      Producer seller = new Producer();
      seller.start();
      Consumer buyer = new Consumer();
      buyer.start();
   }
}
```

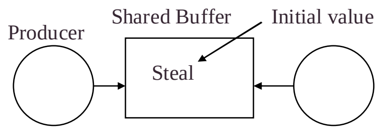
Could also have started *unnamed* threads:

new Producer.**start**();
new Consumer.**start**();

```
Class Producer extends Thread {
   public void run() {
      while(true) {
         System.out.println(" Buy ");
         yield();
      }
   }
}
```

```
Class Consumer extends Thread {
   public void run() {
      while(true) {
         System.out.println(" OK");
         yield();
      }
   }
}
```

Mutually Exclusive Java Threads




```
Public class ProducerConsumer {
    static Object buffer = new Object();
    public static void main (...) {
        Producer seller = new Producer();
        seller.start();
        Consumer buyer = new Consumer();
        buyer.start();
    }
}
```

Need more here, but we will ignore it

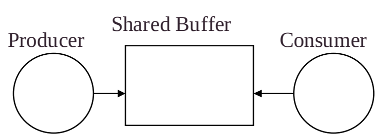
Mutex is OK, but the condition synchronization is wrong!: Output to window *can* be:

No
Buy
OK

```
Class Producer extends Thread {
    public void run() {
        while(true) {
            synchronized (buffer) {
                buffer = "Buy";
                System.out.println(" Buy ");
            }
            yield();
        }
    }
}
```

```
Class Consumer extends Thread {
    public void run() {
        while(true) {
            synchronized (buffer) {
                if (buffer == "Buy") System.out.println(" OK");
                    else System.out.println(" No");
            }
            yield();
        }
    }
}
```

Synchronizing and Mutually Exclusive Java Threads




```
Class Producer extends Thread {
    public void run() {
        while(true) {
            synchronized (buffer) {
                while <full>  wait(nonfull_object);
                buffer = "Buy";
                System.out.println(" Buy ");
                notifyAll(nonempty_object);
            }
            yield();
        }
    }
}
```

```
Class Consumer extends Thread {
    public void run() {
        while(true) {
            synchronized (buffer) {
                while <empty> wait(nonempty_object);
                if (buffer == "Buy") System.out.println(" OK");
                    else System.out.println(" No");
                notifyAll(nonfull_object);
            }
            yield();
        }
    }
}
```

Notify

- No FIFO order when waking!
- Must reevaluate

But stop right there about wait() and notify()

- All is OK in the bounded buffer if the threads are waken up as a result of a notify
- But we can send an interrupt() to a thread and wake it up!
  - Can not Put/Get in this situation, so need something to catch an interrupt from interrupt():
    - **try** {wait();} **catch** (InterruptedException e) {<analyze and take care of the exception e>}
    - In effect we have support for some user level exception handling
    - Will propagate upwards until termination if not handled

Exceptions in Java

Java	Others	In class	Comments
Exception	Exception	Exception. Interrupt	User level releases an exception. HW releases an interrupt.
Throwing	Raising	Releasing	Causing an exception
Catching	Handling	Handling. Trapping.	Trapping an exception and taking care of it
Catch clause	Handler	Trap Handler	The code taking care of the exception
Stack trace	Call chain	Stack call trace	The sequence of (call) statements that brought control to the operation where the exception happened

Java Daemon Threads

setDaemon(boolean on)

- **true**
- **false**

- Serves other threads in an application
- Application exits when there are only daemons left
- Examples
  - timer
  - network socket connections

Size of Java threads

- Each thread default stack size 400Kbytes
- 0.5Kbytes for internal state
- A Unix process: 2Gbyte address space
  - => about 5000 Java threads
  - But other limitations imposed by
    - CPU availability, Swap space, Disk bandwidth
  - Try it (the system will grind to a halt)
- Number of threads needed depend upon application
  - Use threads to achieve concurrency
  - Overlap CPU and I/O

Pthreads

- Portable Operating System Interface (POSIX) threads
- Unix, Windows NT (freeware)
- And no daemon support :-)

Pthread library functions

- pthread_create (thread_ID,…)
- pthread_exit
- pthread_join (thread_ID,...)
- pthread_detach (thread_ID)
- pthread_cancel
- pthread_kill

Mutex and condition synchronization

- Intra process mutex
  - shared by the threads of the process
- Inter process mutex
  - shared by threads in different processes
    - Must map the mutex to memory shared by the processes

Mutex in Pthreads

- Creating a mutex
  - Intra-process:
    - static pthread_mutex_t lockname; */Init value is 0=open*/
- pthread_mutex_init
- pthread_mutex_lock
- pthread_mutex_unlock
- pthread_mutex_trylock
- pthread_mutex_destroy

Condition Synchronization in Pthreads

- Condition variable
  - pthread_cond_t condname = PTHREAD_COND_INITIALIZER;
  - Both intra- and inter process
- pthread_cond_signal (condname)
  - Scheduling policy determines which thread
  - OK with just one consumer and one producer
- pthread_cond_broadcast ()
  - All threads waiting will be notifyed and must reevaluate
    - As with all monitors the MUTEX must first be acquired (automatically)
  - OK when several consumers (and producers)
- pthread_cond_wait (condname, lockname)
  - Automatically opens the mutex on lockname
- pthread_cond_timedwait
  - times out and returns error code

Monitors in C using Pthreads

pthread_mutex_lock (&lock);
  while (<buffer empty>) pthread_cond_**wait** (&nonfull, &lock);
  <update buffer and state variables>;
  pthread_cond_**broadcast** (&nonempty);
pthread_mutex_unlock (&lock)

Start **all** threads waiting.

They will all reevaluate if they can continue

No need to remember UNLOCK in C++ and Java because we can declare a class monitor and } will unlock

Read/Write Locks in Pthreads

- See the Readers and Writers example
- Currently no such predefined locks in Pthreads
- Solaris SPLIT (Solaris to POSIX Interface Layer for Threads) has these locks
  - rwlock_init
  - rw_rdlock and unlock
  - rw_wrlock and unlock

Spin locks in Pthreads

- Lock is closed, and we take 37us to do a wait and block! But then the lock is actually only held for 5us by the other thread! Much time wasted.
- Try a spin lock:
  - pthread_mutex_trylock()

    if (no success after, say, 10 iterations) pthread_mutex_lock()

Trylock takes about 2us

But remember:

•CR must be short (5us in the example)

•Not sensible on a single CPU (why?)

Try it and see what happens: set iteration counter to 0 and measure time vs. grabbing the lock directly

Semaphores in Pthreads

- sem_t s;
- sem_init (&s, 0, 1); /* Init semaphore s to 1)
- sem_wait (&s)
- sem_trywait (&s)
  - if (semaphore = 0) return status code, no block
- sem_post (&s)

0 means intra process

Scheduling of Pthreads

- Each thread has a priority
- Unblocking waiting threads: order is not always guaranteed, depends upon scheduling policy used
- Preemption the norm
- Scheduling by kernel: thread is declared BOUND
- Scheduling “somewhat” by user level: UNBOUND
- Scheduling policy
  - SCHED_OTHER: default (time slice according to priority), no unblocking order guaranteed
  - SCHED_FIFO: next is hghest priority, longest waiting
  - SCHED_RR: FIFO+RR

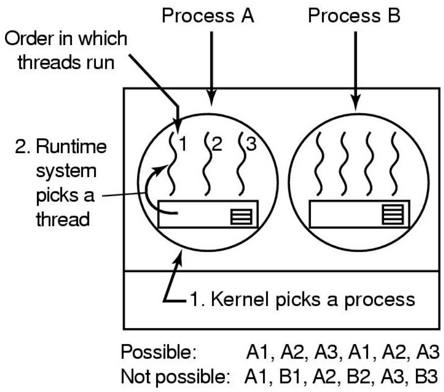
Size of Pthreads

- Solaris default stack size 1MB
  - Thread stacks do not grow automatically!

MT can boost Performance

- Reduce contention to shared data
  - "tiling", more locks, finer granularity of access
  - simpler locks, spin locks
- Reduce overhead
  - One lock instead of several when data items are used together
  - Stuff in inner loops can cost, so remove if possible
- Reduce paging
  - When a thread waits for a page, another one can run
- Communication bandwidth
  - Frequency of synchronization
  - Size of data
- Number of threads: keep all CPUs busy, but not more

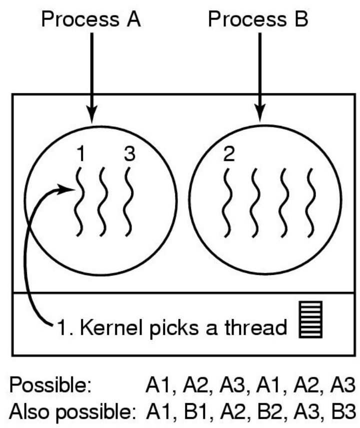
Thread Scheduling (1)



Possible scheduling of user-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

Thread Scheduling (2)



Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst