

Concurrent Programming in Java Thread

Java Thread versus Pthread

Objective

Explain the multithreading paradigm, and all aspects of how to use it in an application

- **Cover basic MT concepts**
- **Review the positions of the major vendors**
- **Contrast POSIX, UI, Win32, and Java threads**
- **Explore (most) all issues related to MT**
- **Look at program and library design issues**
- **Look at thread-specific performance issues**
- **Understand MP hardware design**
- **Examine some POSIX/Java code examples**

At the end of this seminar, you should be able to evaluate the appropriateness of threads to your application, and you should be able to use the documentation from your chosen vendor and start writing MT code.

Background

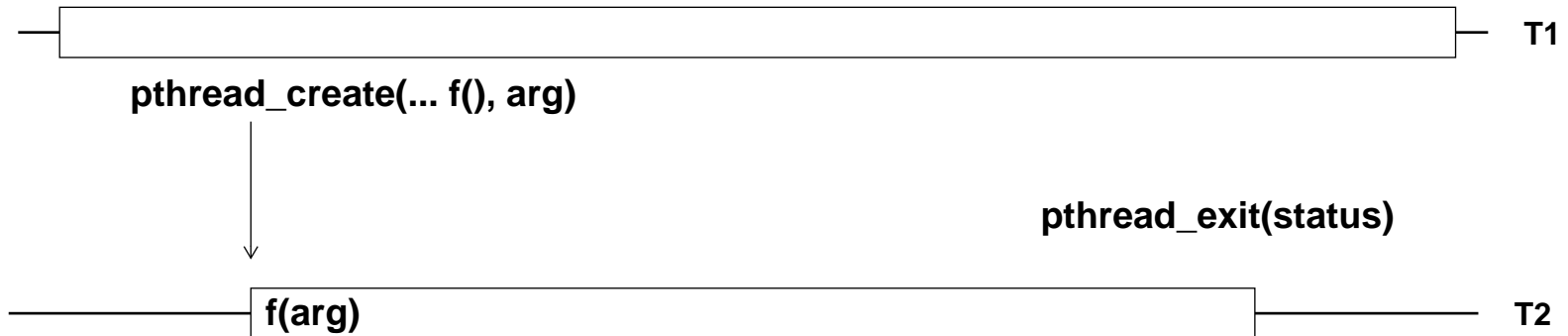
- Experimental threads have been in labs for years
- Many different libraries were experimented with
- Currently there are four major “native” MT libraries in use:
 - POSIX / UNIX98
 - UI (aka “Solaris threads”)
 - OS/2
 - Win32
 - (Apple with Mach)
- POSIX ratified “Pthreads” in June, 1995
- All major UNIX vendors ship Pthreads
- Java Threads (which are usually built on the native threads)

The Value of MT

Threads can:

- Simplify program structure
- Exploit parallelism
- Improve throughput
- Improve responsiveness
- Minimize system resource usage
- Simplify realtime applications
- Simplify signal handling
- Enable distributed objects
- Compile from a single source across platforms (POSIX, Java)
- Run from a single binary for any number of CPUs

Thread Life Cycle



POSIX

```
main()
{...
pthread_create(f, arg)
...
}
```

```
void *f(void *arg)
```

```
{...
pthread_exit(status);
}
```

Win32

```
main()
{...
CreateThread(f, arg)
_beginthread(f, arg)
_xbeginthread(f, arg)
}
```

```
DWORD f(DWORD arg)
```

```
{...
ExitThread(status);
_endthread(status);
_xendthread(status);
}
```

Java

```
main()
{ MyThread t;

  t = new MyThread();
  t.start();
}
```

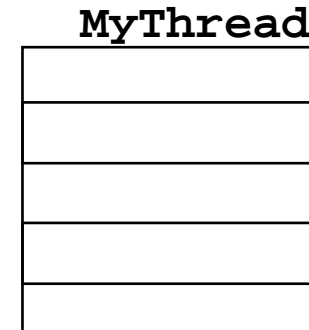
```
class MyThread
  extends Thread
{
  public void run()
  {...
  return()
}
}
```

Runnable vs. Thread

It is possible to subclass `Thread` and define a `run()` method for it:

```
public MyThread extends Thread
{
    public void run()
    {do_stuff();}
}
```

```
MyThread t = new MyThread();
t.start();
```

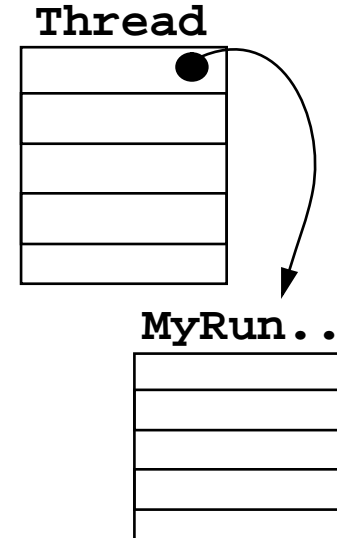


Runnable vs. Thread

But it is (always ?) better to define a `Runnable` and have a stock `Thread` run it:

```
public MyRunnable implements Runnable
{
    public void run()
    {do_stuff();}
}
```

```
Thread t = new Thread(new MyRunnable());
t.start();
```



Logically speaking, we are not changing the nature of the thread, so we shouldn't be subclassing it. Plus the fact, now we can subclass something more useful if we want. (No multiple inheritance in Java.)

Self Starting Threads

It is also possible to have a thread start running as soon as construction is complete:

```
public MyRunnable implements Runnable
{
    public MyRunnable()
    {new Thread(this).start();}

    public void run()
    {do_stuff();}
}
```

But I don't think this gains you anything (you save one line of code) and subclassing gets rather hairy.

Don't do this.

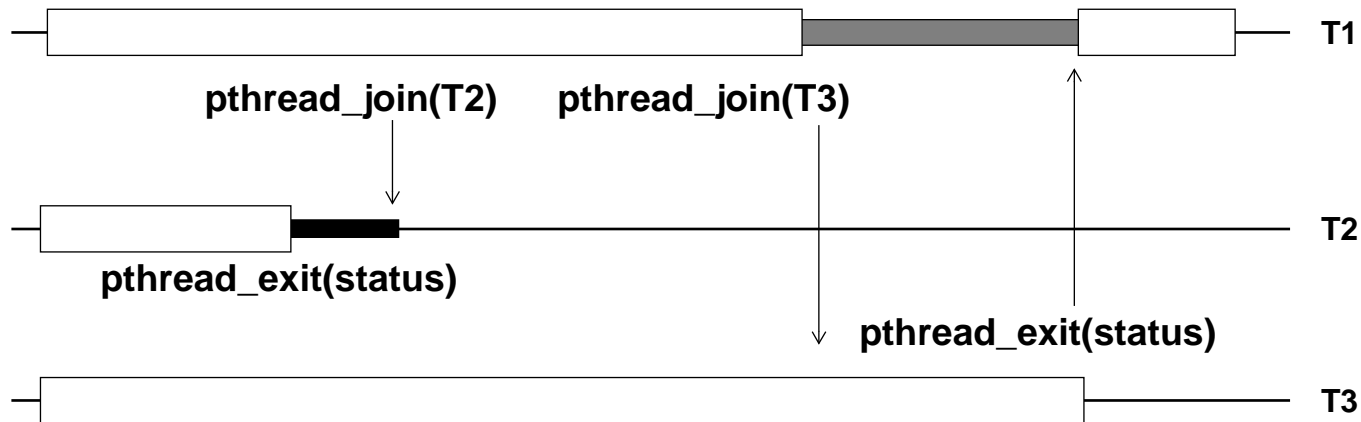
Restarting a Thread

In Java, a `Thread` object is just an object that happens to have a pointer to the actual thread (stack, kernel structure, etc.). Once a thread has exited (“stopped”), it is gone. It is not “restartable” (whatever that means!).

A `Runnable`, on the other hand, may be used for as many threads as you like. Just don’t put any instance variables into your `Runnable`.

We like to view `Threads` as the engines for getting work done, while the `Runnable` is the work to be done.

Waiting for a Thread to Exit



POSIX

Win32

Java

`pthread_join(T2);`

`WaitForSingleObject(T2)`

`T2.join();`

There is no special relation between the creator of a thread and the waiter. The thread may die before or after join is called. You may join a thread only once.

EXIT vs. THREAD_EXIT

The normal C function `exit()` always causes the process to exit. That means all of the process -- All the threads.

The thread exit functions:

POSIX: `pthread_exit()`

Win32: `ExitThread()` and `endthread()`

Java: `thread.stop()` (vs. `System.exit()`)

UI: `thr_exit()`

all cause only the calling thread to exit, leaving the process intact and all of the other threads running. (If no other (non-daemon) threads are running, then `exit()` will be called.)

Returning from the initial function calls the thread exit function implicitly (via “falling off the end” or via `return()`).

Returning from `main()` calls `_exit()` implicitly. (C, C++, not Java)

stop() is Deprecated in JDK 1.2

As called from the thread being stopped, it is equivalent to the thread exit functions. As called from other threads, it is equivalent to POSIX asynchronous cancellation. As it is pretty much impossible to use it correctly, stop has been proclaimed officially undesirable.

If you wish to have the current thread exit, you should have it return (or throw an exception) up to the run() method and exit from there. The idea is that the low level functions shouldn't even know if they're running in their own thread, or in a "main" thread. So they should never exit the thread at all.

destroy() was Never Implemented

So don't use it.

POSIX Thread IDs are *Not* Integers

They are defined to be opaque structures in all of the libraries. This means that they cannot be cast to a `(void *)`, they can not be compared with `==`, and they cannot be printed.

In implementations, they CAN be...

- Solaris: `typedef unsigned int pthread_t`
 - main thread: 1; library threads: 2 & 3; user threads 4, 5...
- IRIX: `typedef unsigned int pthread_t`
 - main thread: 65536; user threads ...
- Digital UNIX: ?
- HP-UX: `typedef struct _tid{int, int, int} pthread_t`

I define a print name in `thread_extensions.c`:

```
thread_name(pthread_self()) -> "T@123"
```

Thread IDs are *Not* Integers

Good Programmer

```
if (pthread_equal(t1, t2)) ...
```

```
int foo(pthread_t tid)
{...}
```

```
foo(pthread_self());
```

```
pthread_t tid = pthread_self();
```

```
printf("%s", thread_name(tid));
```

??!

Bad Programmer

```
if (t1 == t2) ...
```

```
int foo(void *arg)
{...}
```

```
foo((void *) pthread_self());
```

```
printf("t%d", tid);
```

```
#define THREAD_RWLOCK_INITIALIZER \
    {PTHREAD_MUTEX_INITIALIZER, \
     NULL_TID}
```

Java Thread Names

A Java Thread is an Object, hence it has a `toString()` method which provides an identifiable, printable name for it.

You may give a thread your own name if you so wish:

```
Thread t1 = new Thread("Your Thread");
```

```
Thread t2 = new Thread(MyRunnable, "My Thread");
```

```
System.out.println(t2 + " is running.");
```

```
==> Thread[My Thread,5,] is running.
```

```
System.out.println(t2.getName() + " is running.");
```

```
==> My Thread is running.
```

`sched_yield()`

`Thread.yield()`

- Will relinquish the LWP (or CPU for bound threads) to anyone who wants it (at the same priority level, both bound and unbound).
- Must NOT be used for correctness!
- Probably not useful for anything but the oddest programs (and then only for “efficiency” or “fairness”).
- The JVM on some platforms (e.g., Solaris with Green threads) may require it under some circumstances, such as computationally bound threads.) This will change.
- Never Wrong, but...

Avoid `sched_yield()` `Thread.yield()` !

Dæmon Threads

A dæmon is a normal thread in every respect save one: Dæmons are not counted when deciding to exit. Once the last non-dæmon thread has exited, the entire application exits, taking the dæmons with it.

Dæmon Threads exist only in UI, and Java threads.

UI:

```
thr_create(..., THR_DAEMON, ...);
```

Java:

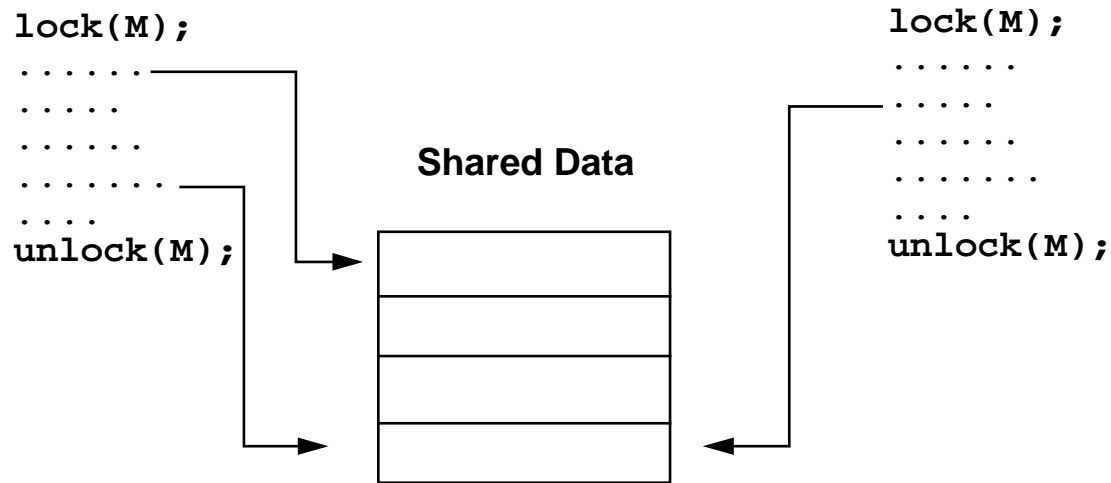
```
void thread.setDaemon(boolean on);
```

```
boolean thread.isDaemon();
```

Avoid using Dæmons!

Synchronization

Critical Sections (Good Programmer)



(Of course you must know *which* lock protects *which* data!)

Synchronization Variables

Each of the libraries implement a set of “synchronization variables” which allow different threads to coordinate activities. The actual variables are just normal structures in normal memory*. The functions that operate on them have a little bit of magic to them.

UI: **Mutexes, Counting Semaphores, Reader/Writer Locks, Condition Variables, and Join.**

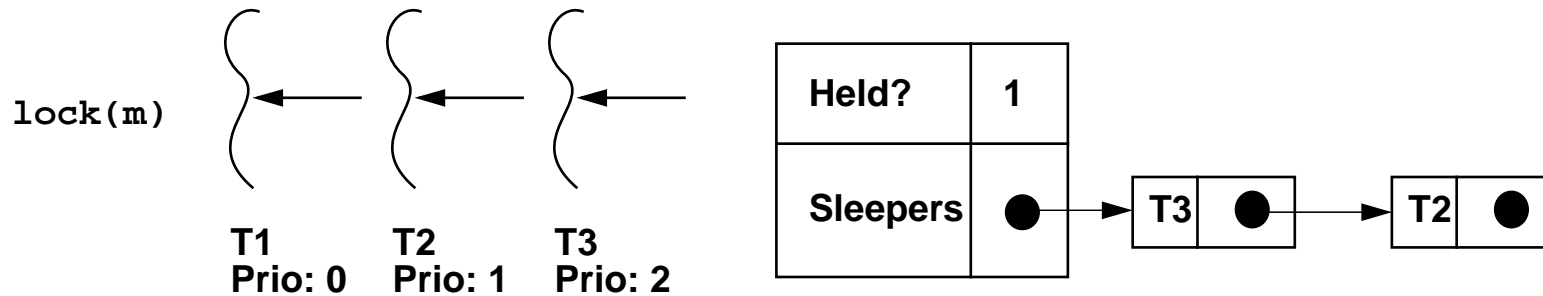
POSIX: **Mutexes, Counting Semaphores, Condition Variables, and Join.**

Java: **Synchronized, Wait/Notify, and Join.**

Win32: **Mutexes, Event Semaphores, Counting Semaphores, “Critical Sections.” and Join (WaitForObject).**

*** In an early machine, SGI actually built a special memory area.**

Mutexes



POSIX

```
pthread_mutex_lock(&m)
...
pthread_mutex_unlock(&m)
```

Win32

```
WaitForSingleObject(m)
...
ReleaseMutex(m)
```

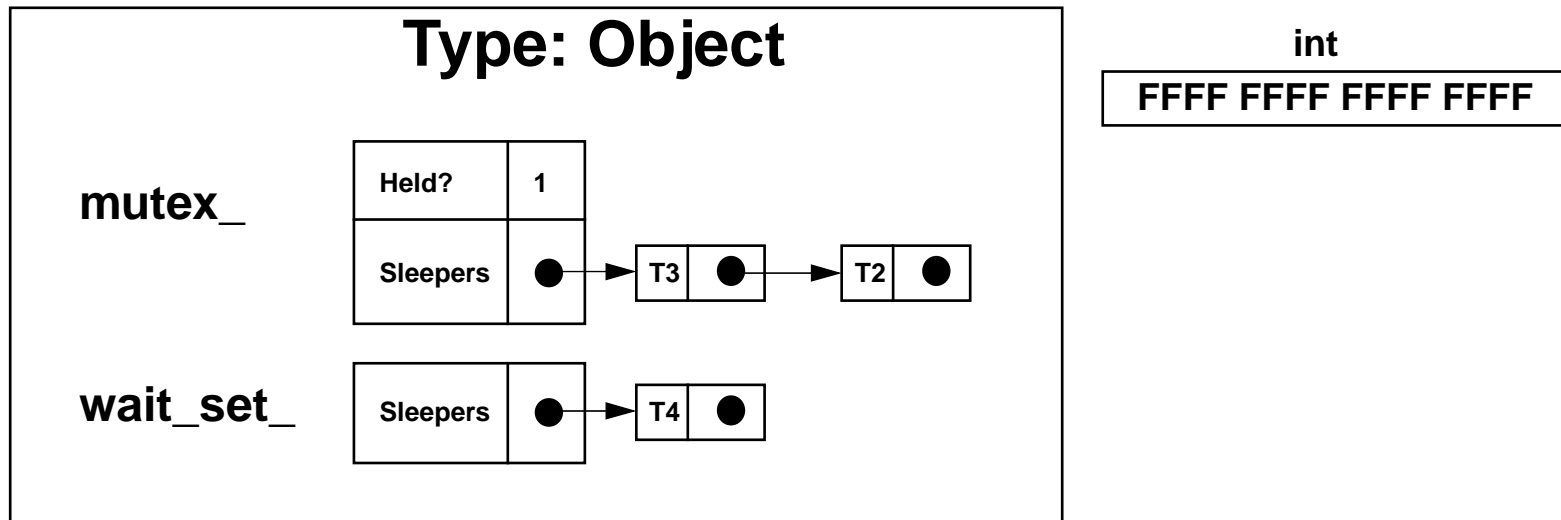
Java

```
synchronized(obj)
{ ...
}
```

- **POSIX and UI: Owner not recorded, Block in priority order, Illegal unlock not checked at runtime.**
- **Win32: Owner recorded, Block in FIFO order**
- **Java: Owner recorded (not available), No defined blocking order, Illegal Unlock impossible**

Java Locking

The class Object (and thus everything that subclasses it -- e.g., everything except for the primitive types: int, char, etc.) have a hidden mutex and a hidden “wait set”:



Java Locking

The hidden mutex is manipulated by use of the `synchronized` keyword:

```
public MyClass() {  
    int count=0;  
  
    public synchronized void increment()  
    {count++;}  
  
}
```

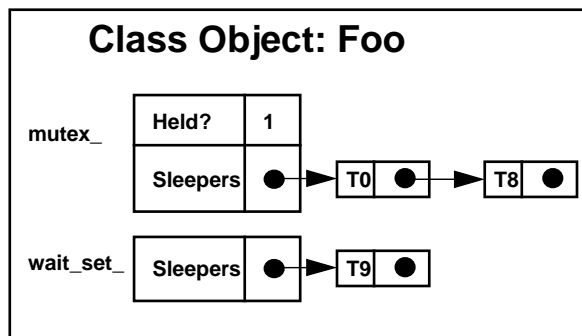
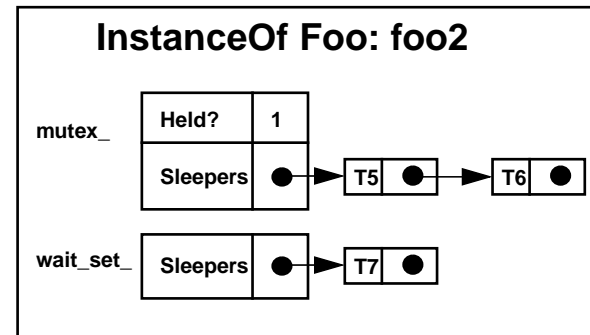
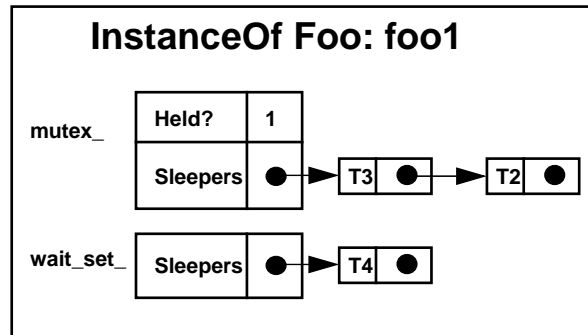
or explicitly using the object:

```
public MyClass() {  
    int count=0;  
  
    public void increment() {  
        synchronized (this)  
            {count++;}  
    }  
  
}
```

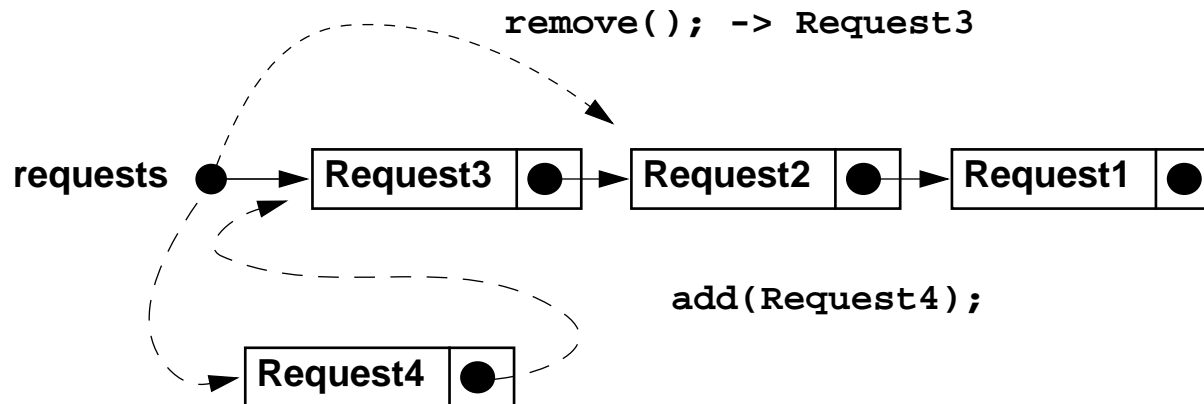
Thus in Java, you don't have to worry about unlocking. That's done for you! Even if the thread throws an exception or exits.

Each Instance Has Its Own Lock

Thus each instance has its own lock and wait set which can be used to protect.... anything you want to protect! (Normally you'll be protecting data in the current object.) Class Objects are Objects...



Using Mutexes



Thread 1

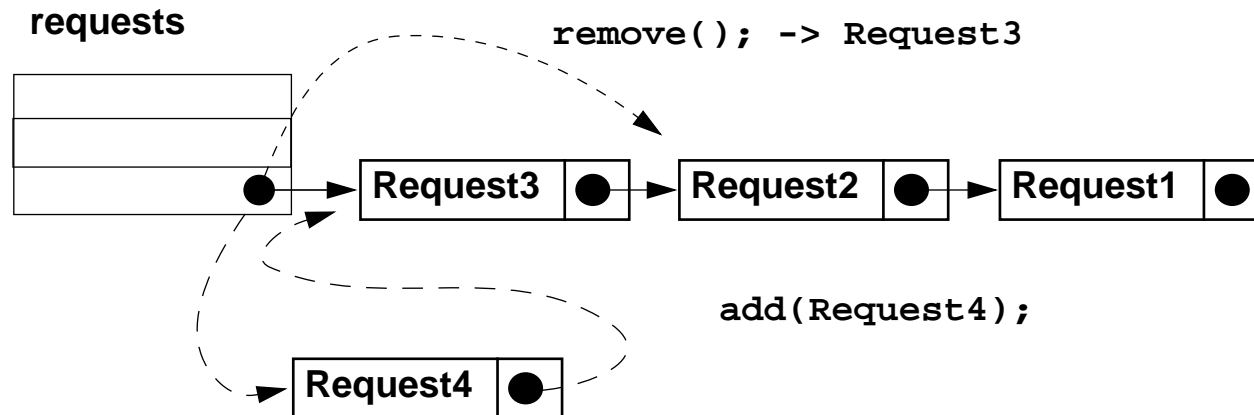
```
add(request_t *request)
{ pthread_mutex_lock(&req_lock);
  request->next = requests;
  requests = request;
  pthread_mutex_unlock(&req_lock)
}
```

Thread 2

```
request_t *remove()
{ pthread_mutex_lock(&req_lock);
  ...sleeping...

  request = requests;
  requests = requests->next;
  pthread_mutex_unlock(&req_lock);
  return(request);
}
```

Using Java Locking



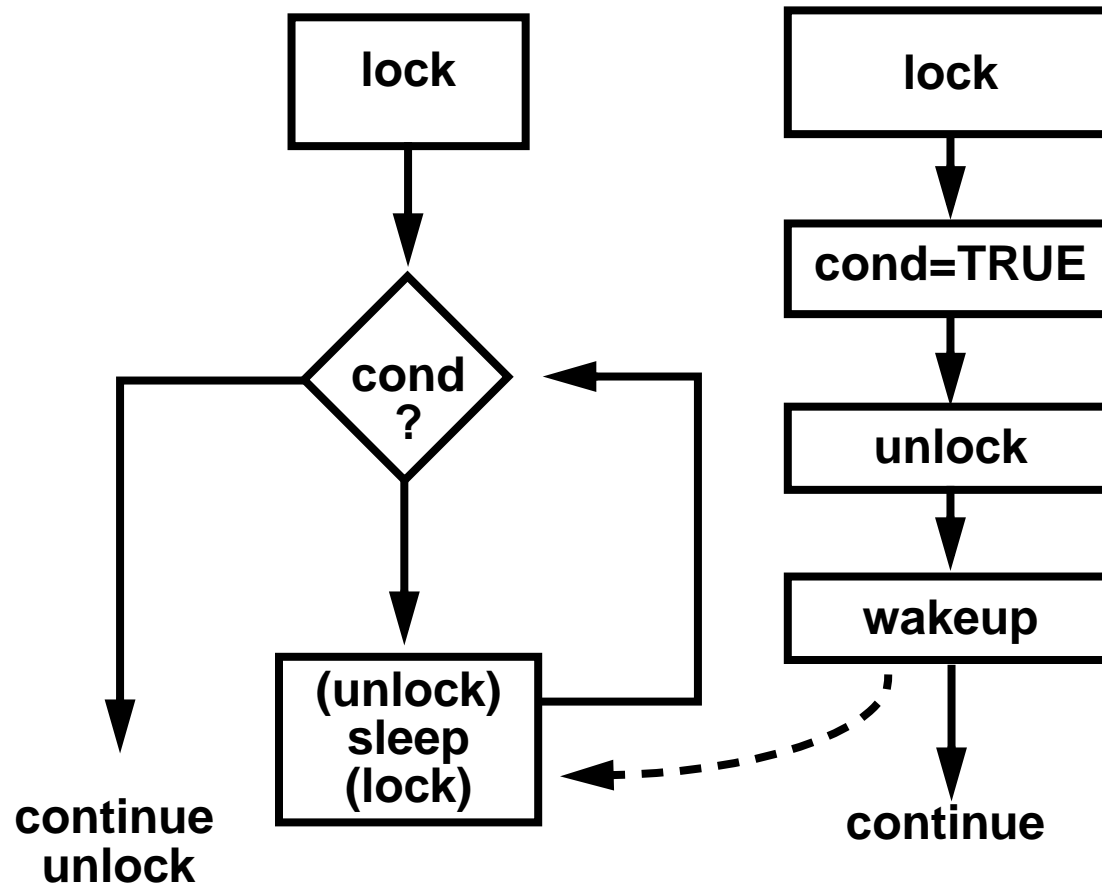
Thread 1

```
add(Request request) {  
    synchronized(requests)  
    {request.next = requests;  
     requests = request;  
    }  
}
```

Thread 2

```
Request remove() {  
    synchronized(requests)  
    ...sleeping...  
  
    {request = requests;  
     requests = requests.next;  
    }  
    return(request);  
}
```

Generalized Condition Variable



Win32 has “EventObjects”, which are similar.

Condition Variable Code

Thread 1

```
pthread_mutex_lock(&m);  
while (!my_condition)  
    pthread_cond_wait(&c, &m);  
  
do_thing();  
pthread_mutex_unlock(&m);
```

Thread 2

```
pthread_mutex_lock(&m);  
my_condition = TRUE;  
pthread_mutex_unlock(&m);  
pthread_cond_signal(&c);  
/* pthread_cond_broadcast(&c); */
```

Java wait() / notify() Code

Thread 1	Thread 2
<pre>synchronized (object) {while (!object.my_condition) object.wait(); do_thing(); }</pre>	<pre>synchronized (object); {object.my_condition = true; object.notify(); // object.notifyAll(); }</pre>

(Explicit use of synchronization)

Java wait() / notify() Code

Thread 1	Thread 2
<pre>public synchronized void foo() {while (!my_condition) wait(); do_thing(); }</pre>	<pre>public synchronized void bar() { my_condition = true; notify(); /* notifyAll();*/ }</pre>

(Implicit use of synchronization)

You can actually combine the explicit and implicit uses of synchronization in the same code.

Nested Locks Are Not Released

If you already own a lock when you enter a critical section which you'll be calling `pthread_cond_wait()` or Java `wait()` from, you are responsible for dealing with those locks:

```
public synchronized void foo()  
{  
    synchronized(that)  
        {while (!that.test()) that.wait();}  
}
```

```
pthread_mutex_lock(&lock1);  
pthread_mutex_lock(&lock2);  
while (!test()) pthread_cond_wait(&cv, &lock2);
```

Java Exceptions for `wait()`, etc.

A variety of Java threads methods throw `InterruptedException`. In particular: `join()`, `wait()`, `sleep()`. The intention is all blocking functions will throw `InterruptedException`.

This exception is thrown by our thread when another thread calls `interrupt()` on it. The idea is that these methods should be able to be forced to return should something external affect their usefulness.

As of Java 1.1, `thread.interrupt()` was not implemented. Anyway, all of our code must catch that exception:

```
try {
    while (true) {
        item = server.get();
        synchronized (workpile) {
            workpile.add(item);
            workpile.wait();
        }
    }
} catch (InterruptedException ie) {}
```

Inserting Items Onto a List (Java)

```
public class Consumer implements Runnable {
public void run() {
    try {
        while (true) {
            synchronized (workpile) {
                while (workpile.empty()) workpile.wait();
                request = workpile.remove();
            }
            server.process(request);
        } }
    catch (InterruptedException e) {} // Ignore for now
}
```

```
public class Producer implements Runnable {
public void run() {
    while (true) {
        request = server.get();
        synchronized (workpile) {
            workpile.add(request);
            workpile.notify();
        } }
}
```

Implementing Semaphores in Java

```
public class Semaphore
{int count = 0;

    public Semaphore(int i)
    {count = i;}

    public Semaphore()
    {count = 0;}

    public synchronized void init(int i)
    {count = i;}

    public synchronized void semWait()
    {while (count == 0)
        {try
            wait();
            catch (InterruptedException e) {}
        }
        count--;}

    public synchronized void semPost()
    {
        count++;
        notify();
    }
}
```

Synchronization Variable Prototypes

```
pthread_mutex_t
error pthread_mutex_init(&mutex, &attribute);
error pthread_mutex_destroy(&mutex);
error pthread_mutex_lock(&mutex);
error pthread_mutex_unlock(&mutex);
error pthread_mutex_trylock(&mutex);

pthread_cond_t
error pthread_cond_init(&cv, &attribute);
error pthread_cond_destroy(&cv);
error pthread_cond_wait(&cv, &mutex);
error pthread_cond_timedwait(&cv, &mutex, &timestruct);
error pthread_cond_signal(&cv);
error pthread_cond_broadcast(&cv);

sem_t
error sem_init(&semaphore, sharedp, initial_value);
error sem_destroy(&semaphore);
error sem_wait(&semaphore);
error sem_post(&semaphore);
error sem_trywait(&semaphore);

thread_rwlock_t
error thread_rwlock_init(&rwlock, &attribute);
error thread_rwlock_destroy(&rwlock);
error thread_rwlock_rdlock(&rwlock);
error thread_rwlock_tryrdlock(&rwlock);
error thread_rwlock_wrlock(&rwlock);
error thread_rwlock_trywrlock(&rwlock);
error thread_rwlock_unlock(&rwlock);
```

Java Synchronization Prototypes

```
synchronized void foo() {...}
synchronized (object) {...}
synchronized void foo() {... wait(); ...}
synchronized void foo() {... notify[All](); ...}
synchronized (object) {... object.wait(); ...}
synchronized (object) {... object.notify[All](); ...}
```

From Bil's Extensions package

```
public void mutex.lock();
public void mutex.unlock();
public void mutex.trylock();

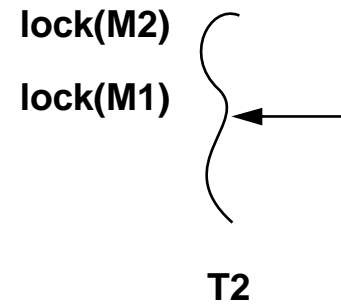
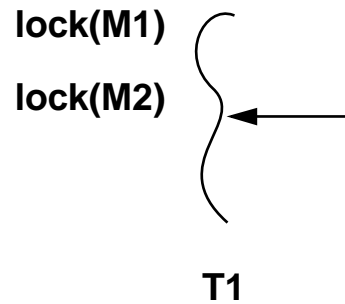
public void cond_wait(mutex);
public void cond_timedwait(mutex, long milliseconds);
public void cond_signal();
public void cond_broadcast();

public void sem_init(initial_value);
public void sem_wait();
public void sem_post();
public void sem_trywait();

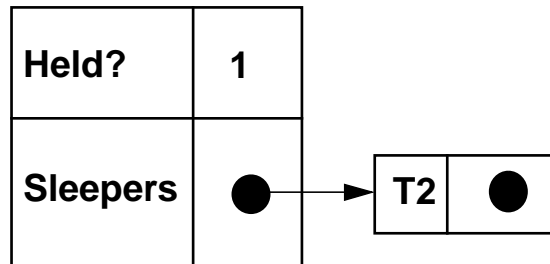
public void rw_rdlock();
public void rw_tryrdlock();
public void rw_wrlock();
public void rw_trywrlock();
public void rw_unlock();
```

*Synchronization
Problems*

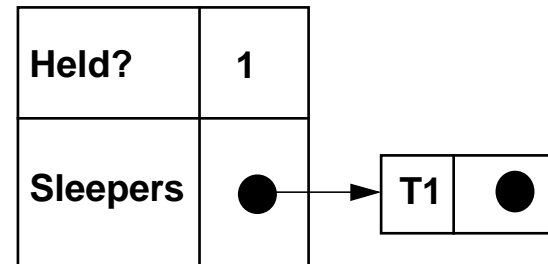
Deadlocks



Mutex M1



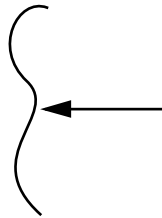
Mutex M2



Unlock order of mutexes cannot cause a deadlock

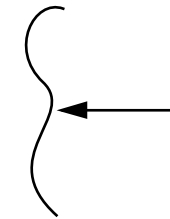
Java Deadlocks

synchronized(M1)
synchronized(M2)



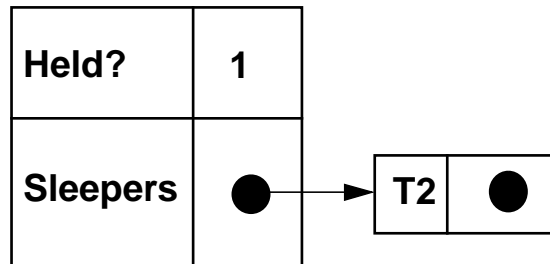
T1

synchronized(M2)
synchronized(M1)

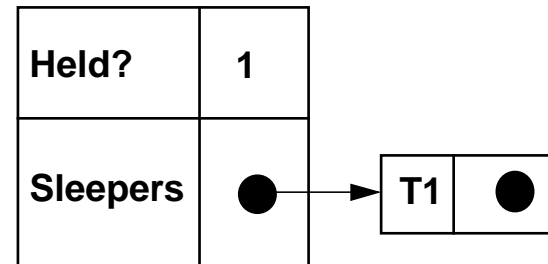


T2

Object M1



Object M2



InterruptedException

Like cancellation, InterruptedException is difficult to handle correctly. You may wish to handle it locally:

```
void foo()  
{  
    try {wait();}  
    catch (InterruptedException e) {do_something}  
}
```

or, more likely, you may wish to simply propogate it up the call stack to a higher level. (Very likely to the very top!)

```
void foo() throws InterruptedException  
{  
    try {wait();}  
}
```

InterruptedException

In any case, it is important not to drop an interruption. Your code may be used by some other package someday...

Bad Programmer:

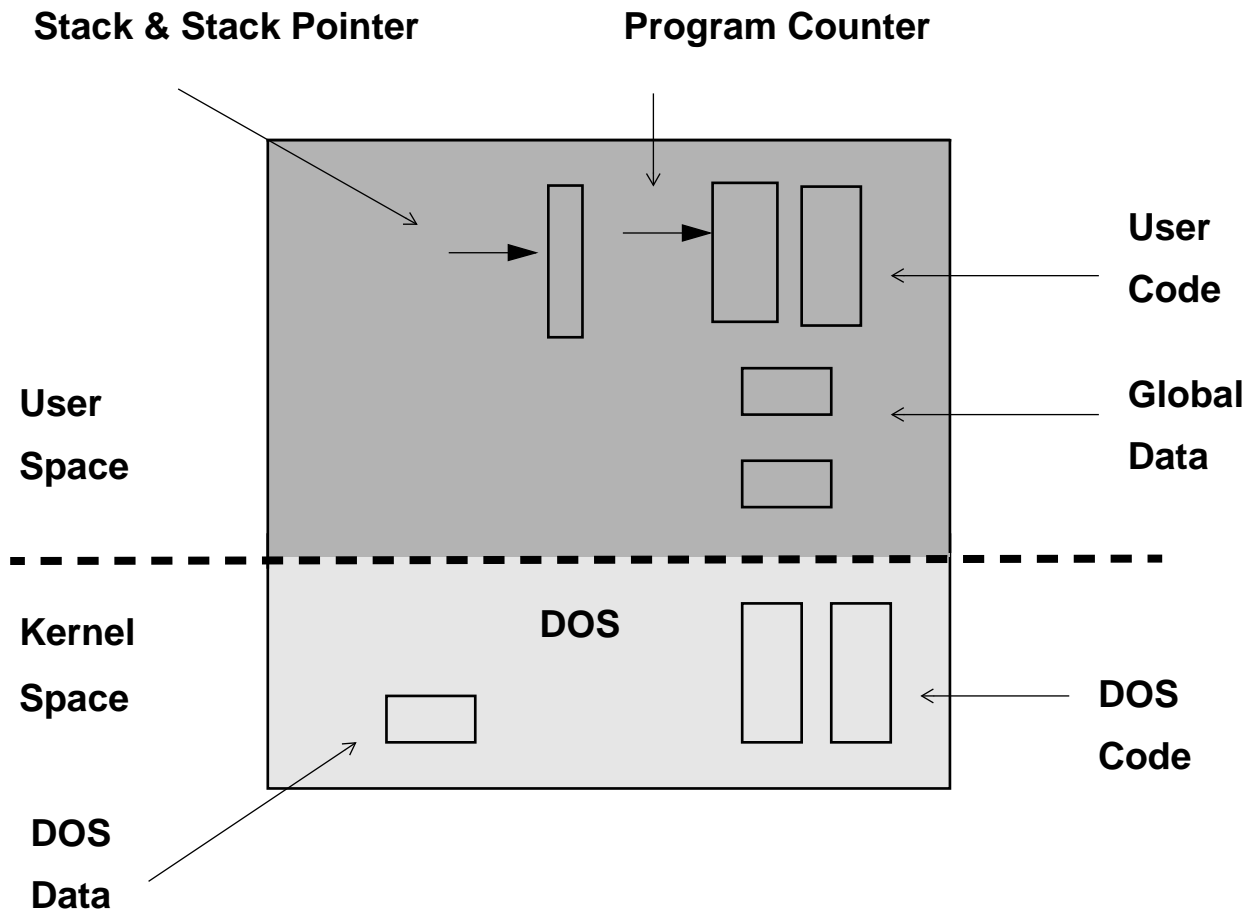
```
void foo() {
    try {wait();}
    catch (InterruptedException e) {}
}
```

Good Programmer:

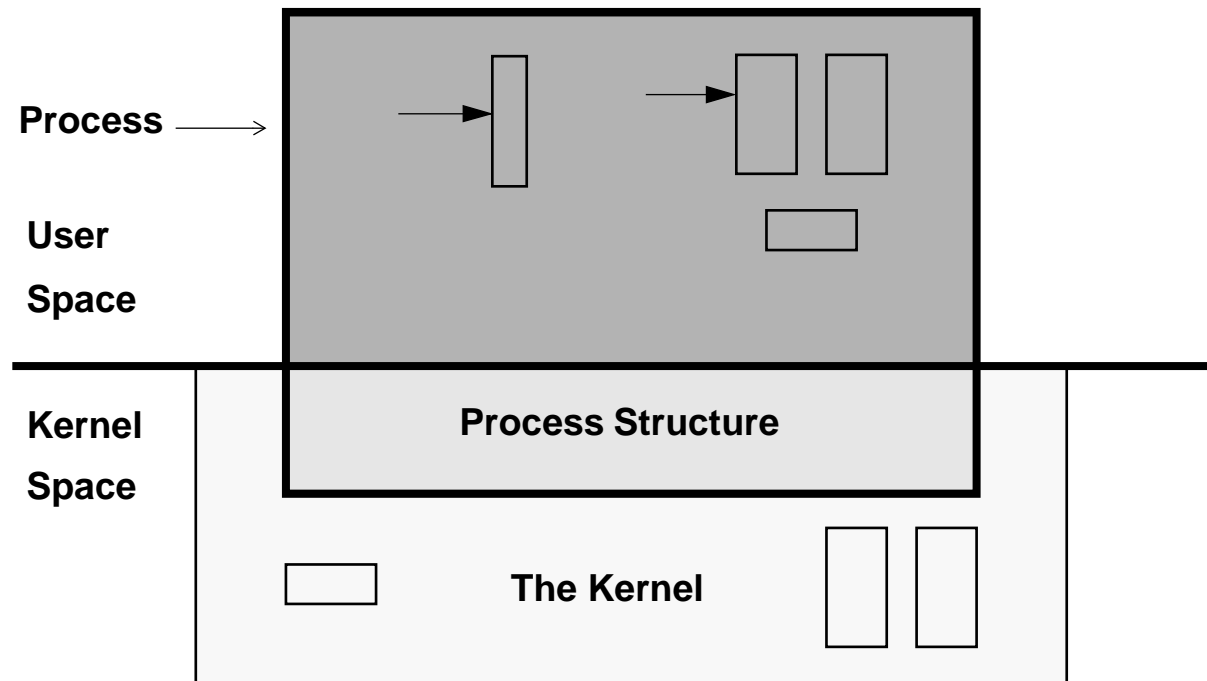
```
void foo() {
    while (true) {
        try {wait(); return;}
        catch (InterruptedException e) {intd=true}
    }
    if (intd) Thread.currentThread().interrupt();
}
```

Foundations

DOS - The Minimal OS



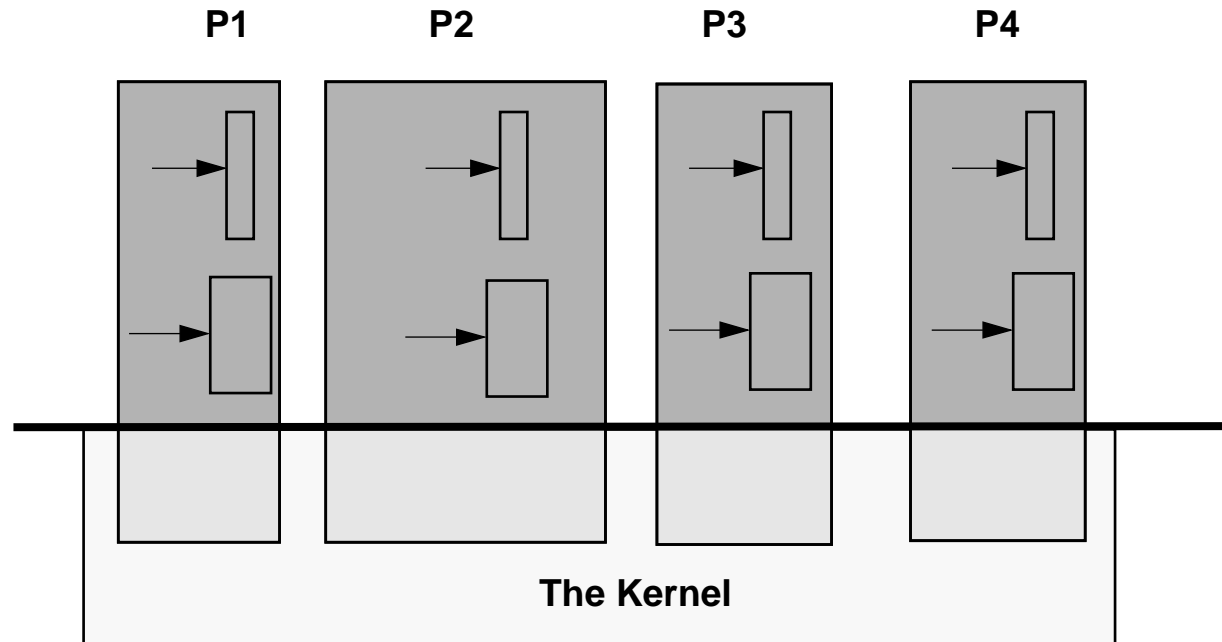
Multitasking OSs



(UNIX, VMS, MVS, NT, OS/2, etc.)

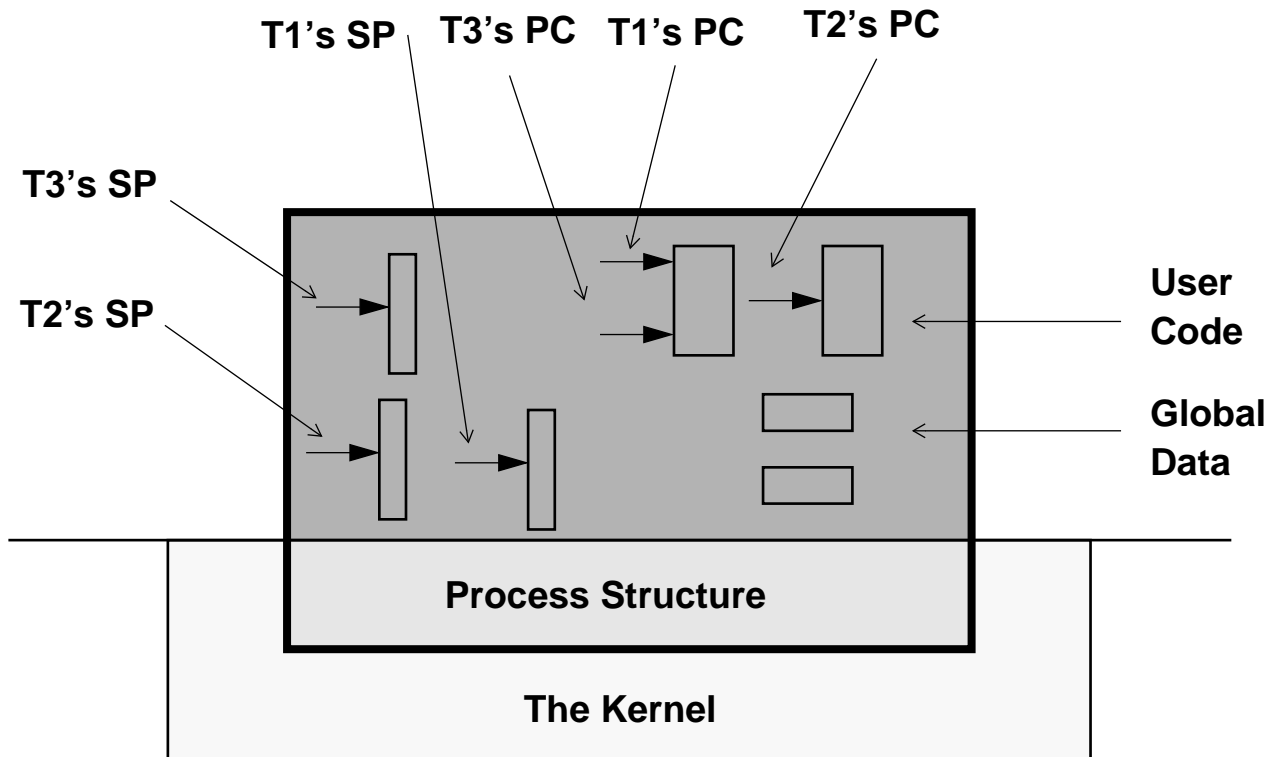
Multitasking OSs

Processes



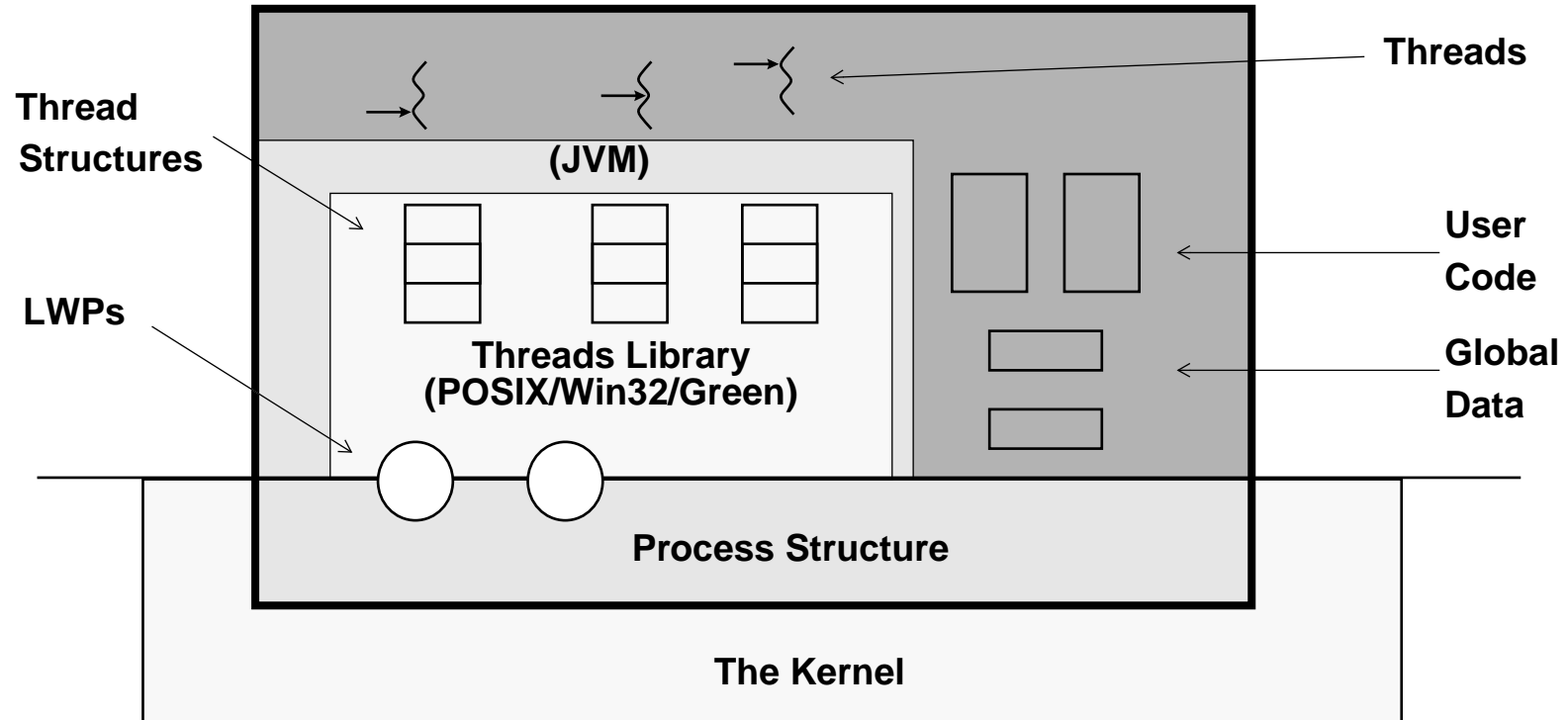
(Each process is completely independent)

Multithreaded Process



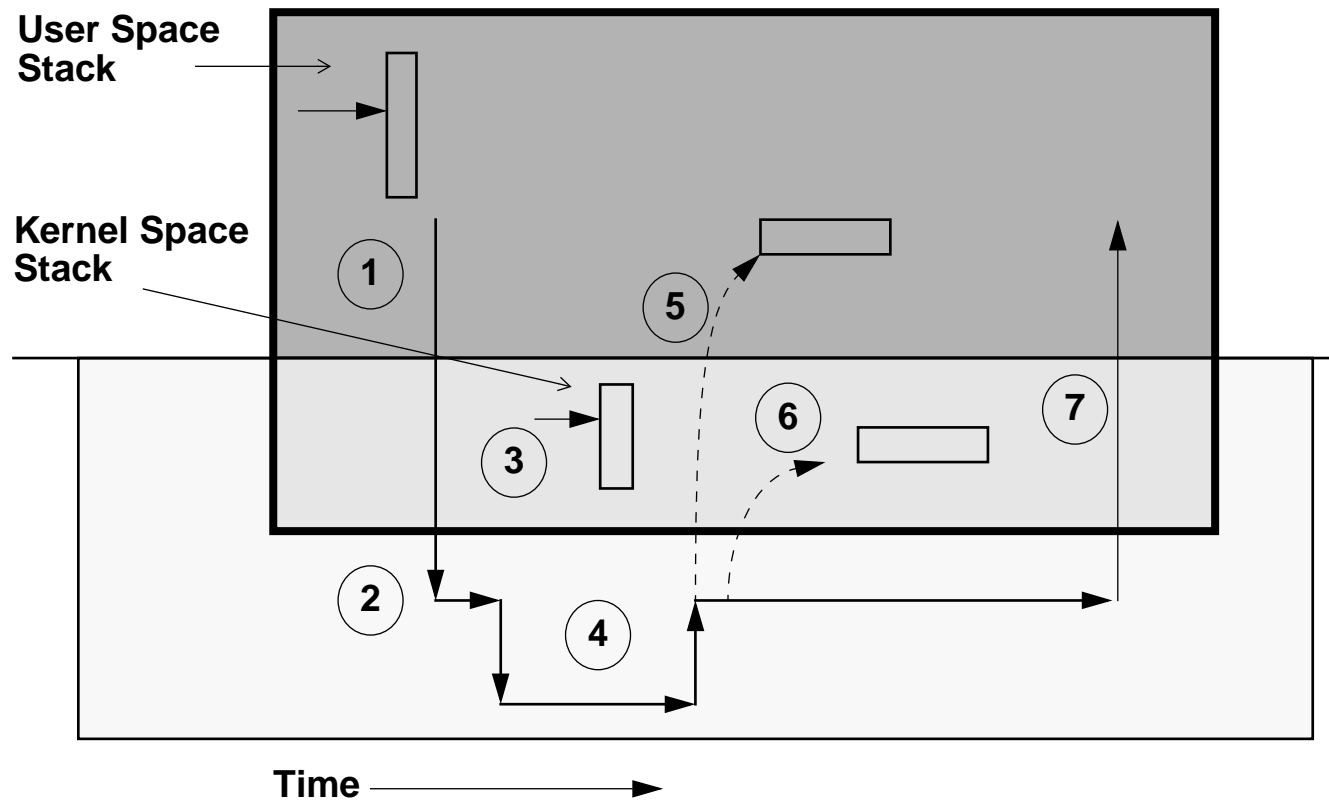
(Kernel state and address space are shared)

Solaris Implementation of Pthreads



libpthread.so is just a normal, user library!
Java threads are part of the JavaVM and similar.

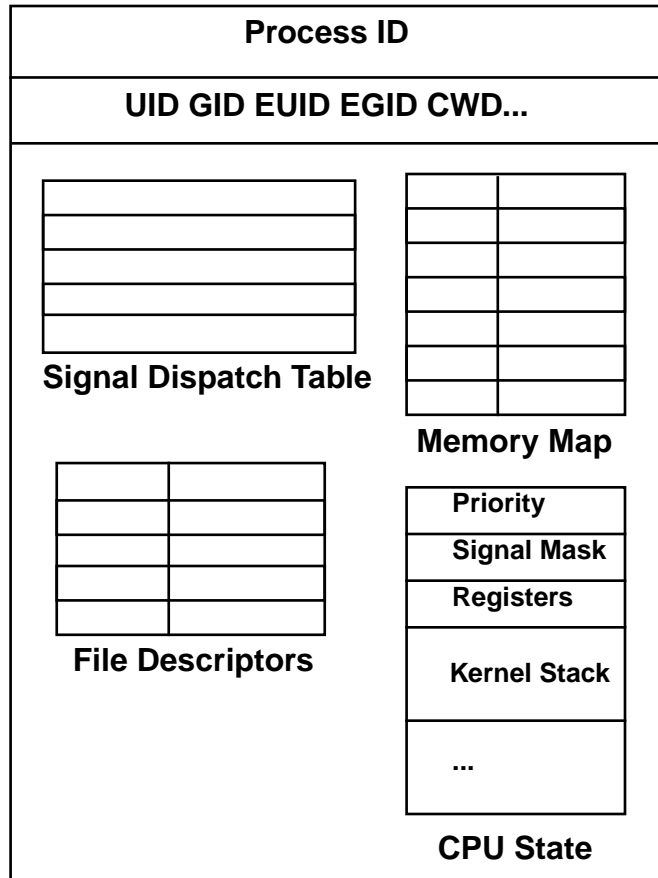
How System Calls Work



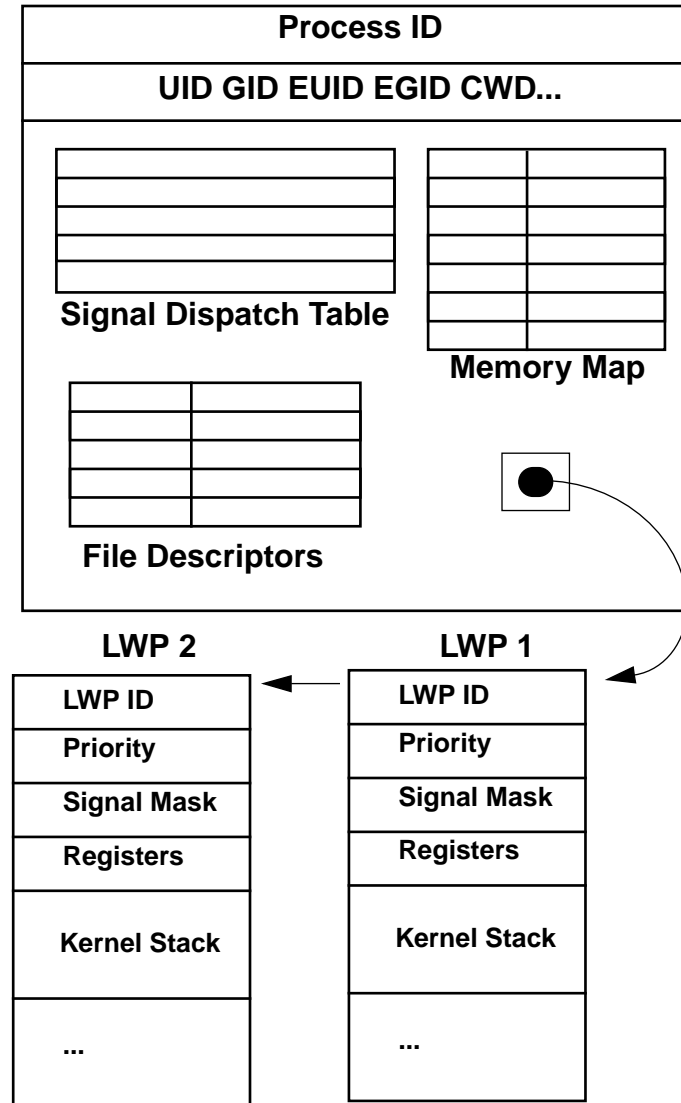
Scheduling

Kernel Structures

Traditional UNIX Process Structure



Solaris 2 Process Structure

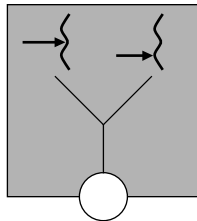


Light Weight Processes

- Virtual CPU
- Scheduled by Kernel
- Independent System Calls, Page Faults, Kernel Statistics, Scheduling Classes
- Can run in Parallel
- LWPs are an *Implementation Technique* (i.e., think about *concurrency*, not LWPs)
- *LWP* is a Solaris term, other vendors have different names for the same concept:
 - DEC: Lightweight threads vs. heavyweight threads
 - Kernel threads vs. user threads (also see “kernel threads” below)

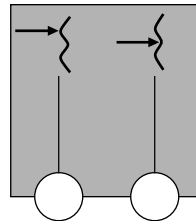
(SUNOS 4.1 had a library called the “LWP” library. No relation to Solaris LWPs)

Scheduling Design Options



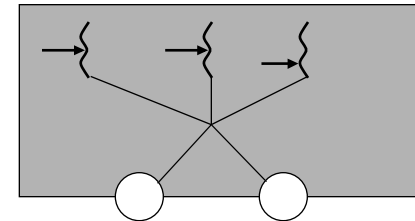
M:1

**HP-UX 10.20
(via DCE)
Green Threads**

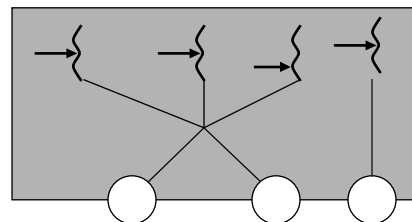


1:1

Win32, OS/2, AIX 4.0



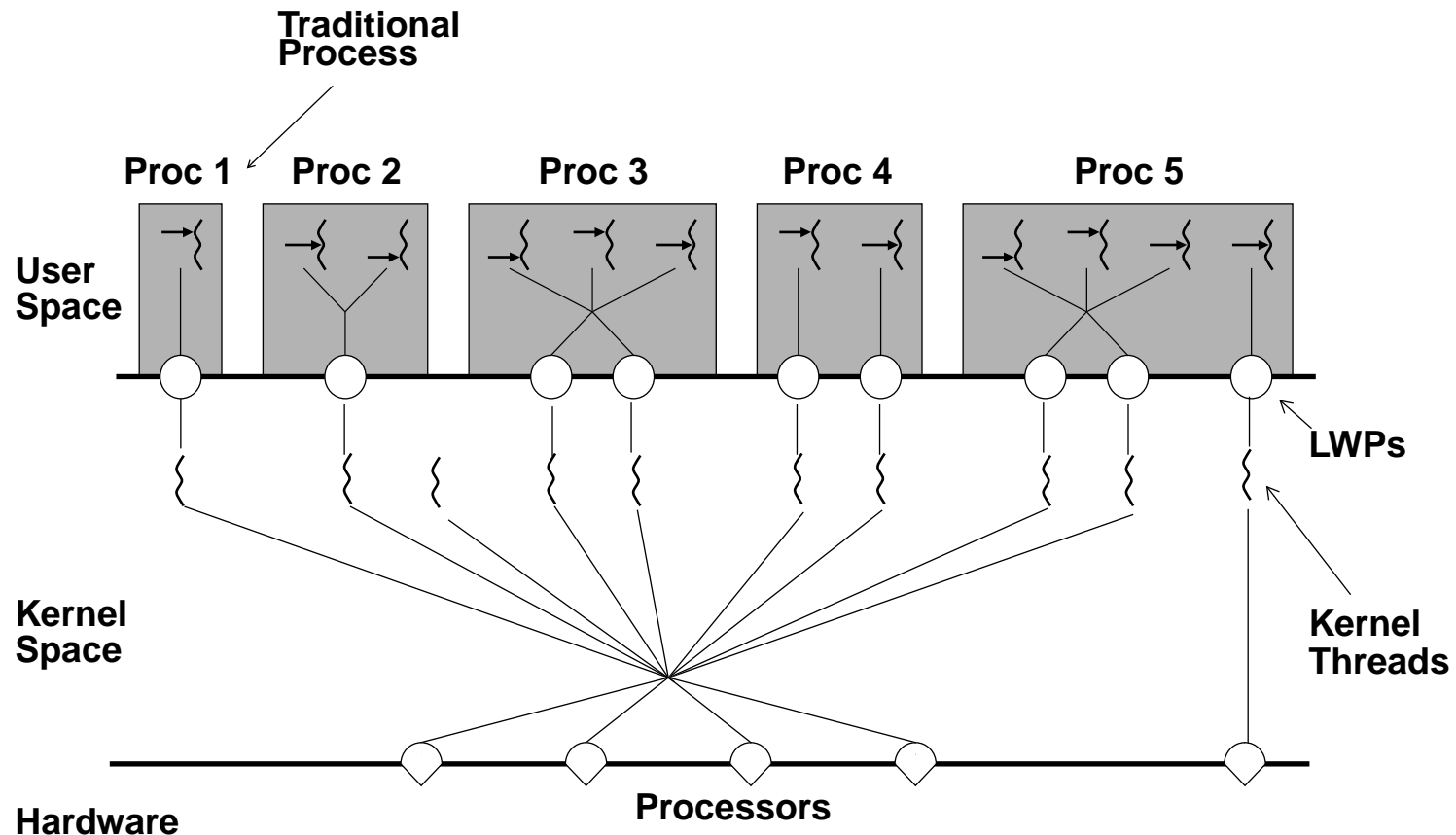
M:M (strict)



2-Level (aka: M:M)

Solaris, DEC, IRIX, HP-UX 10.30, AIX 4.1

Solaris Two-Level Model

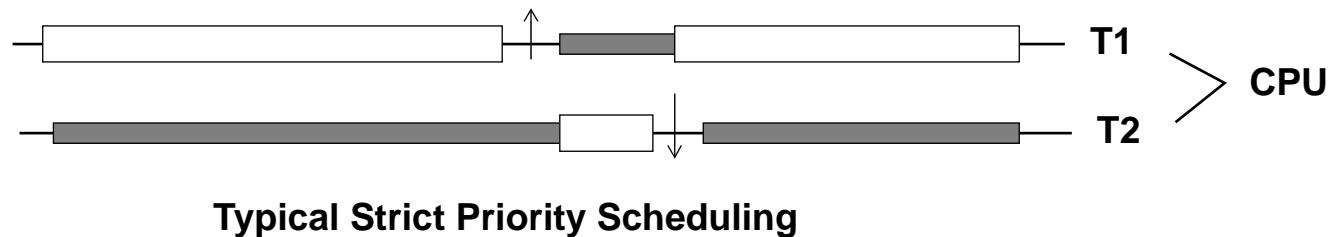
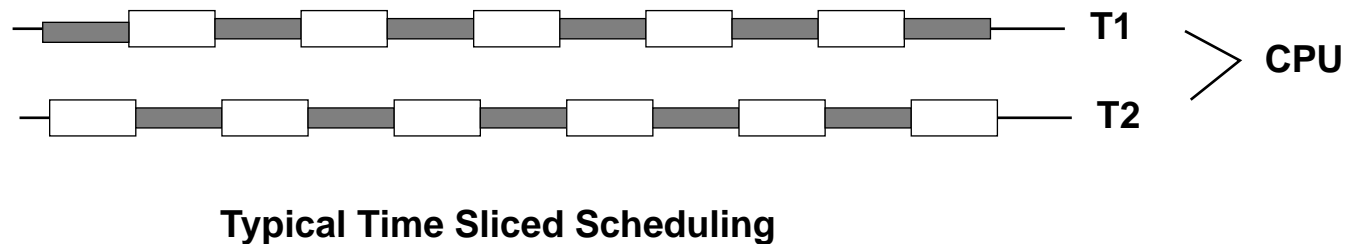


“Kernel thread” here means the threads that the OS is built with.
On HP, etc. it means LWP.

Time Sliced Scheduling

VS.


Strict Priority Scheduling




Working


Sleeping


Request


Reply

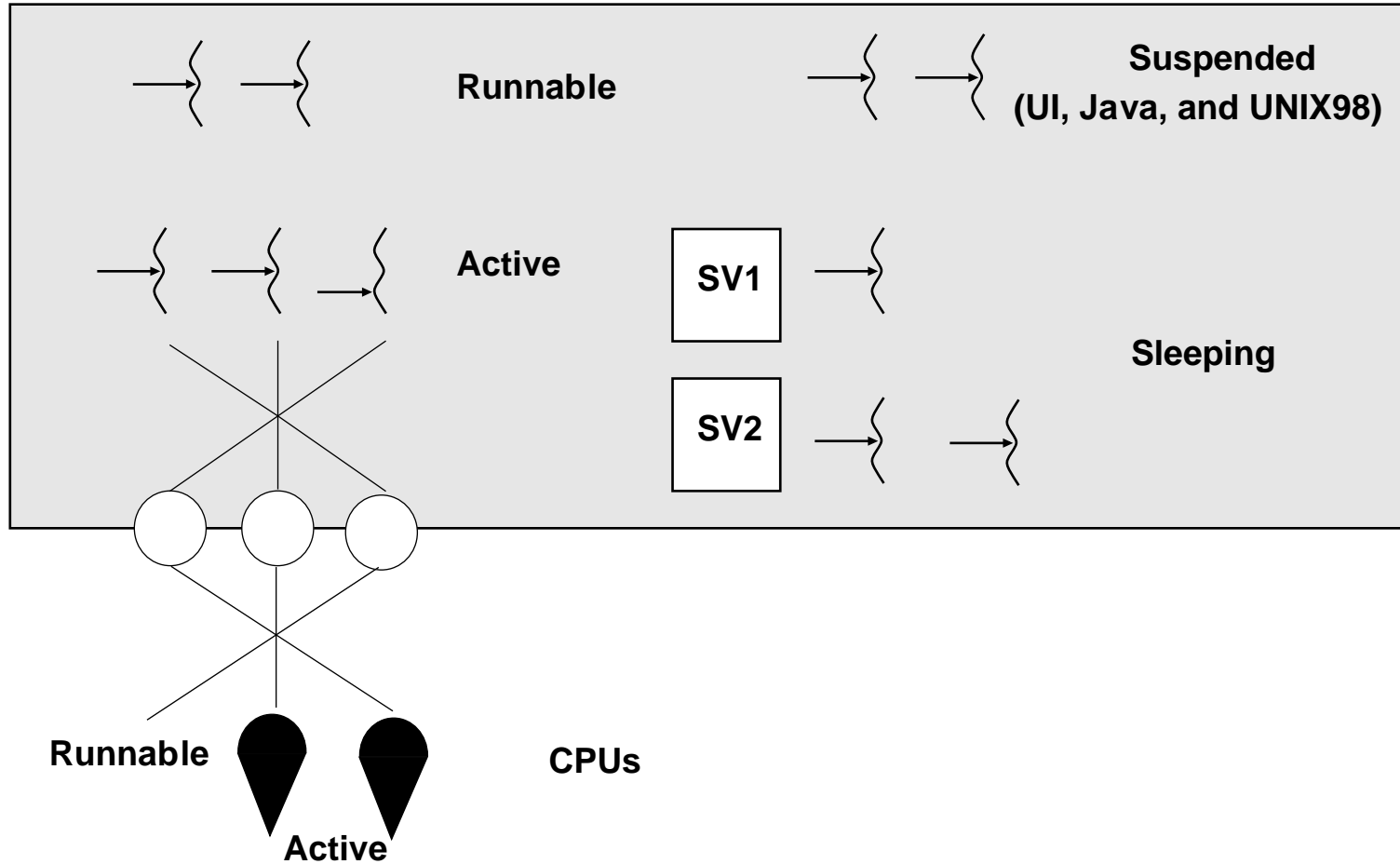
System Scope “Global” Scheduling

VS.

Process Scope “Local” Scheduling

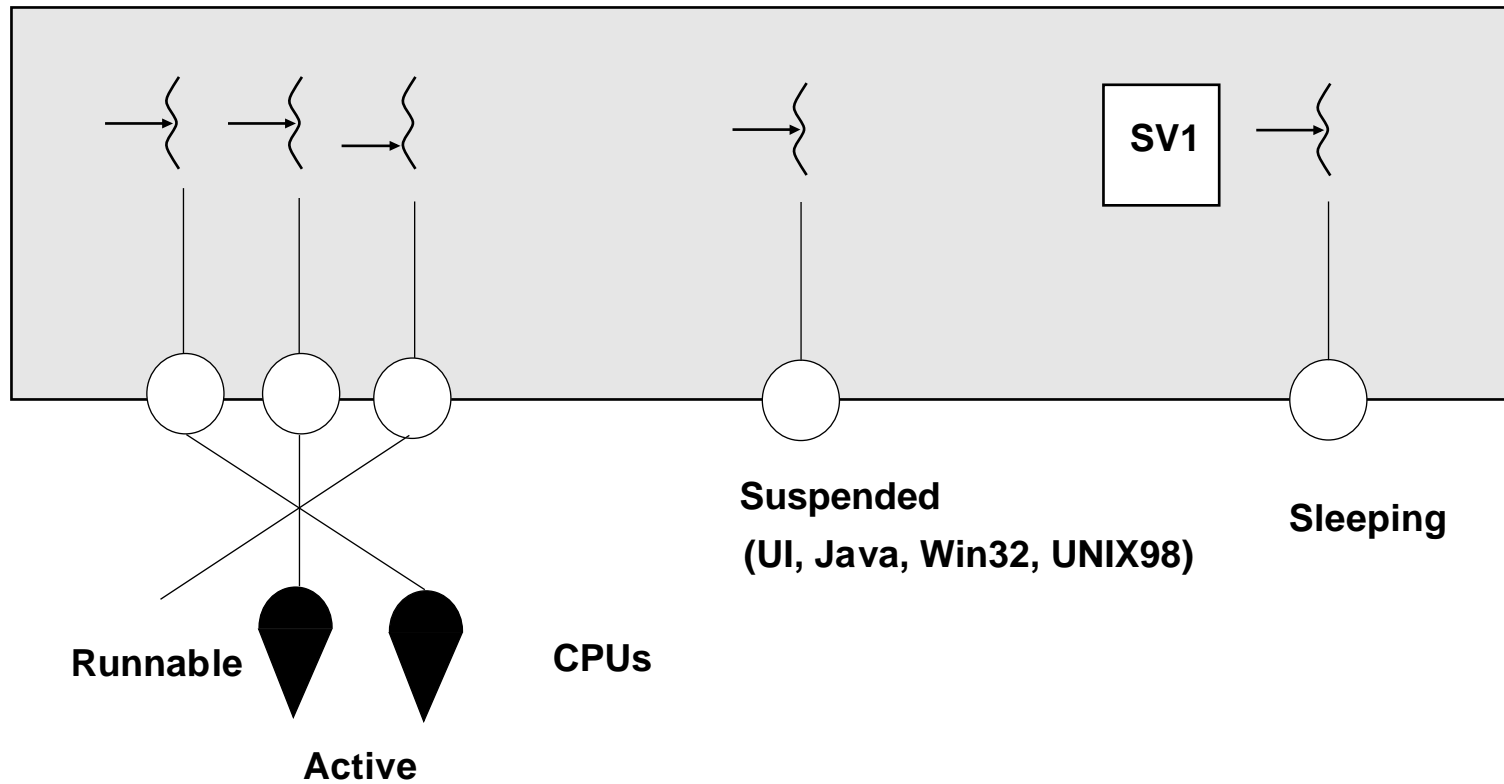
- **Global means that the kernel is scheduling threads (aka “bound threads”, 1:1, System Scope)**
 - **Requires significant overhead**
 - **Allows time slicing**
 - **Allows real-time scheduling**
- **Local means the library is scheduling threads (aka “unbound threads”, M:M, Process Scope)**
 - **Very fast**
 - **Strict priority only, no time slicing (time slicing is done on Digital UNIX, not Solaris)**

Local Scheduling States



(UI, Java, and POSIX only)

Global Scheduling States



(UI, Java, POSIX, Win32)

Scheduler Performance

Creation Primitives

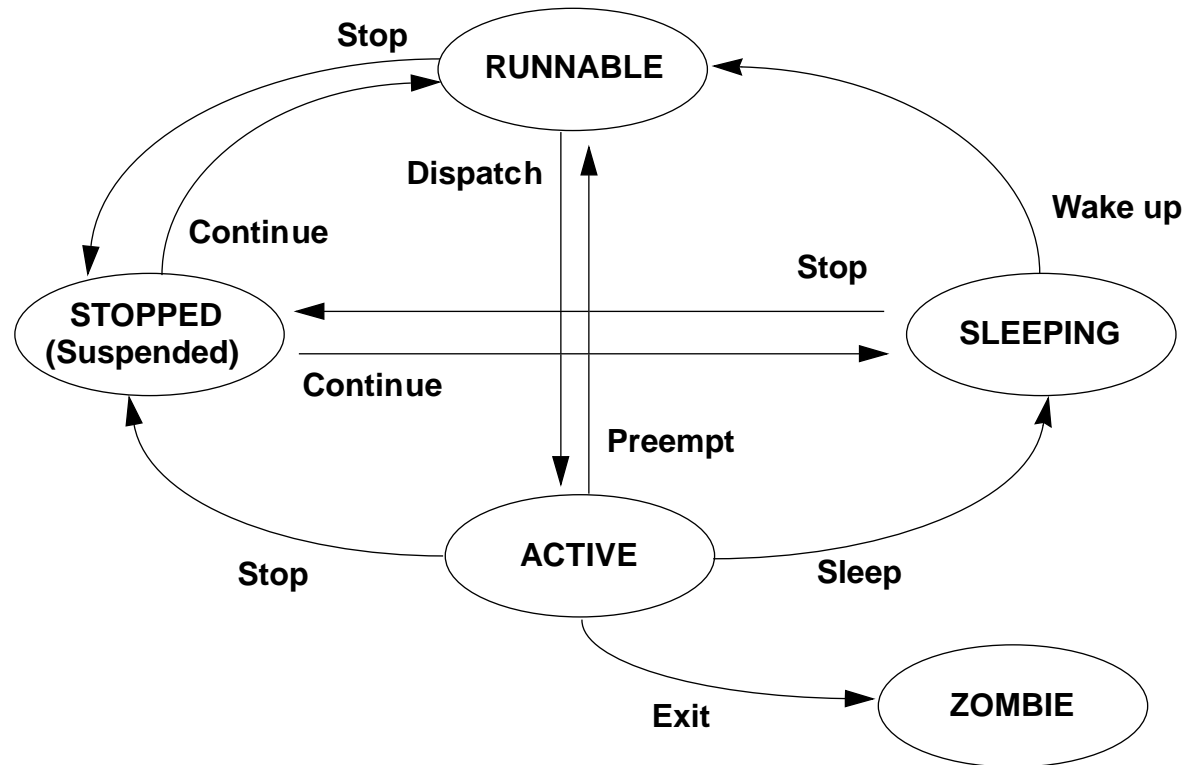
	POSIX	Java 2
	uSecs	uSecs
Local Thread	330	2,600
Global Thread	720	n/a
Fork	45,000	

Context Switching

	POSIX	Java 2
	uSecs	uSecs
Local Thread	90	125
Global Thread	40	n/a
Fork	50	

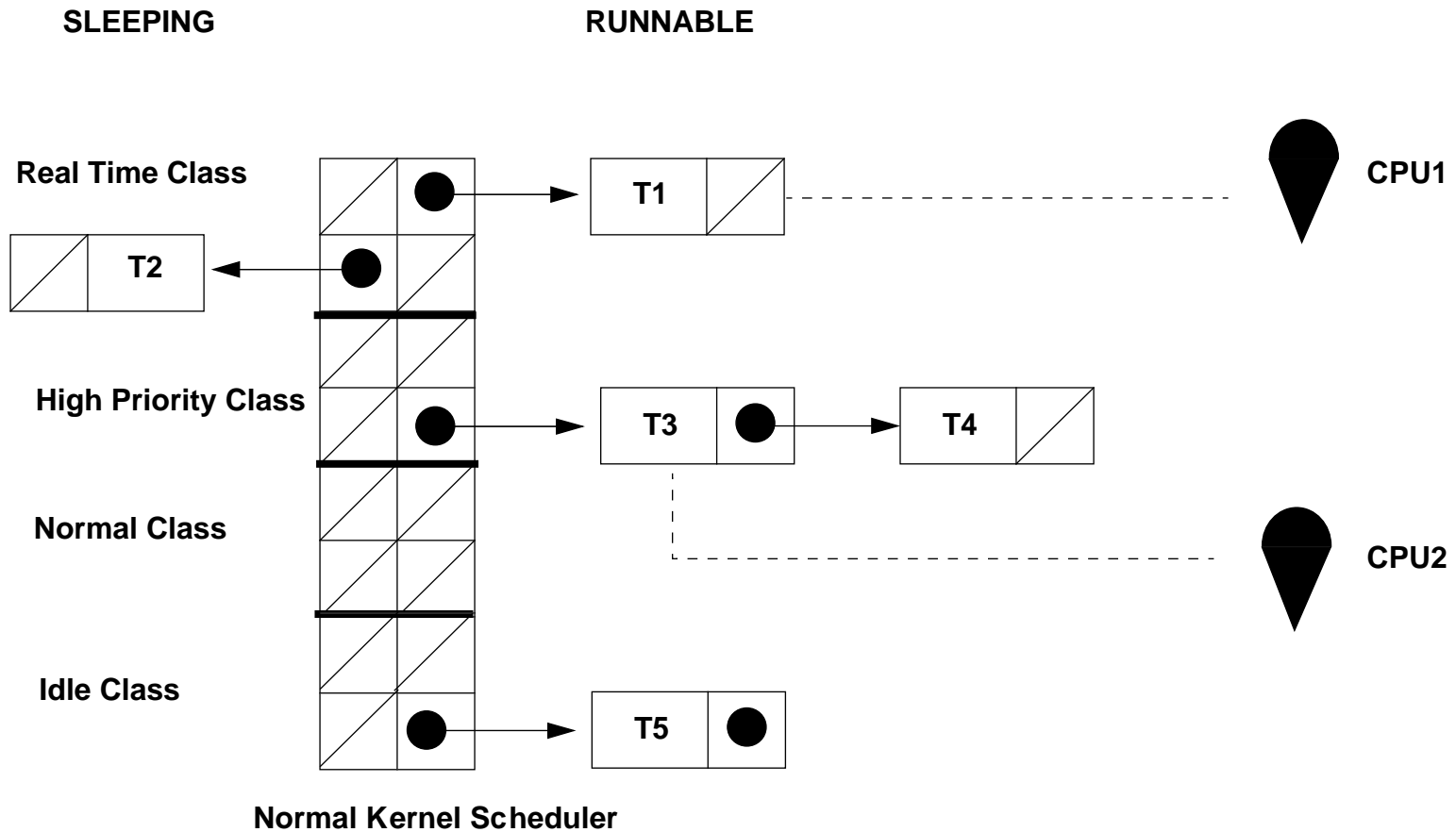
110MHz SPARCstation 4 (This machine has a SPECint of about 20% of the fastest workstations today.) running Solaris 2.5.1.

Scheduling States



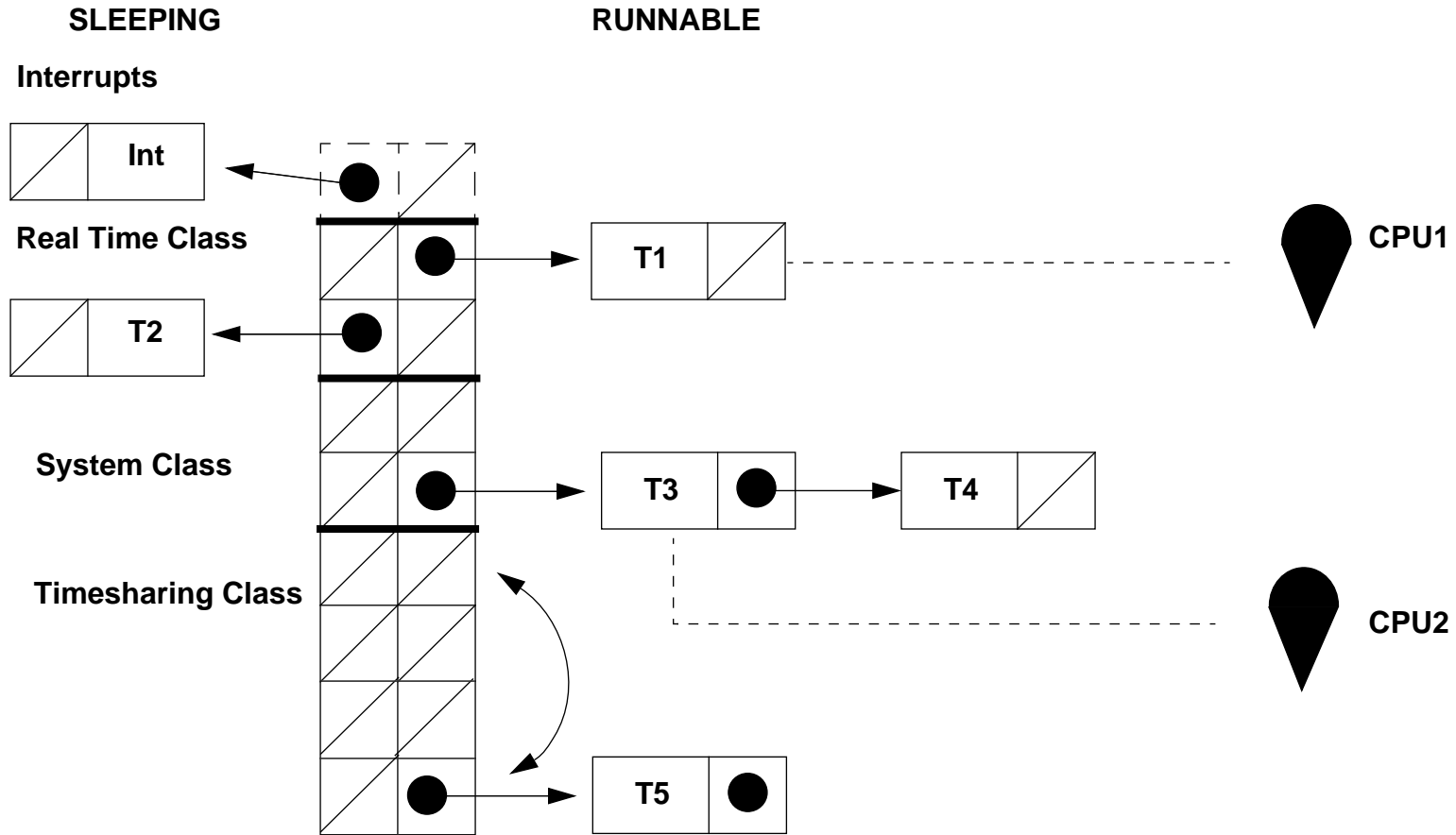
“Stopped” (aka “Suspended”) is only in Win32, Java, and UI threads, not POSIX. No relation to the java method `thread.stop()`

NT Scheduling



All Round Robin (timeslicing)

Solaris Scheduling



Normal Solaris Kernel Scheduler (60 priority levels in each class)

RT and system classes do RR, TS does demotion/promotion.

POSIX Priorities

For realtime threads, POSIX requires that at least 32 levels be provided and that the highest priority threads always get the CPUs (as per previous slides).

For non-realtime threads, the meaning of the priority levels is not well-defined and left to the discretion of the individual vendor. On Solaris with bound threads, the priority numbers are ignored.

You will probably never use priorities.

Java Priorities

The Java spec requires that 10 levels be provided, the actual values being integers between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`. The default level for a new thread is `Thread.NORM_PRIORITY`.

Threads may examine or change their own or others' priority level with `thread.getPriority()` and `thread.setPriority(int)`. Obviously, `int` has to be within the range `Thread.MIN_PRIORITY`, `Thread.MAX_PRIORITY`. It may be further restricted by the thread's group via `threadGroup.getMaxPriority()` and `threadGroup.setMaxPriority(int)`.

On Solaris these are mapped to 1, 10, and 5, although this is not required. These numbers are not propagated down to the LWPs and are therefore meaningless when there are enough LWPs. For Green threads, the priority levels are absolute.

You will probably never use priorities.