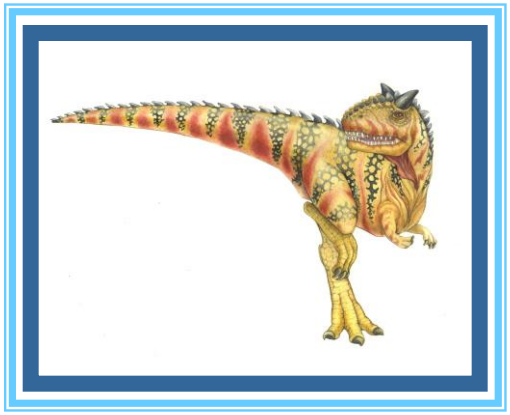
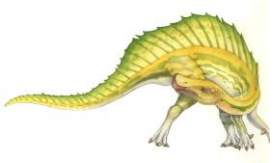
Chapter 4: Threads



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

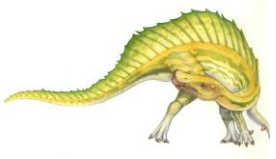


Objectives

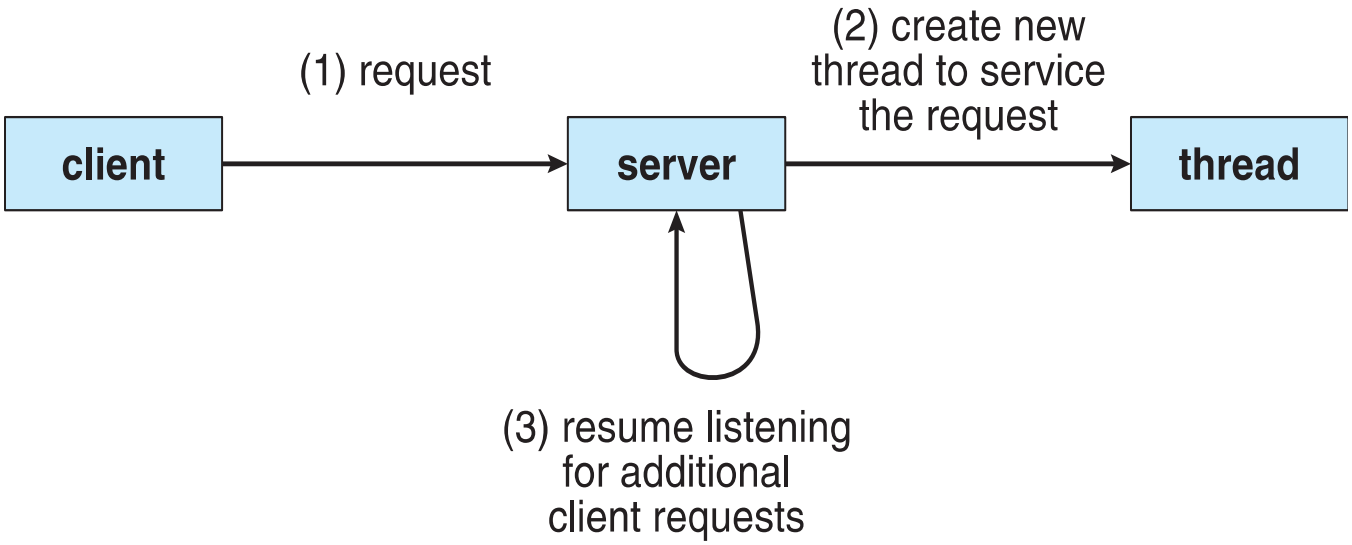
- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

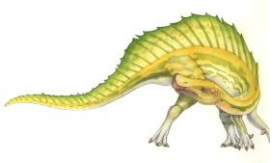
Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded



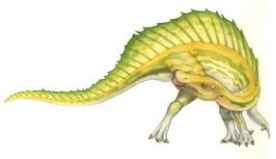
Multithreaded Server Architecture





Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

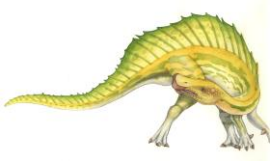


Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

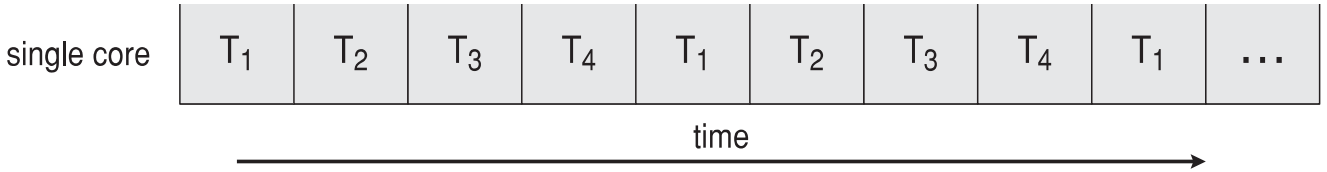
Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

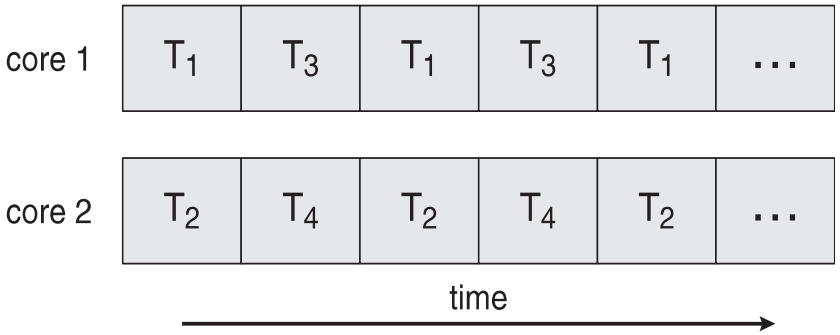


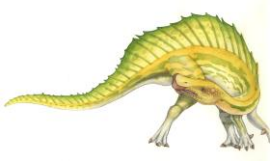
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

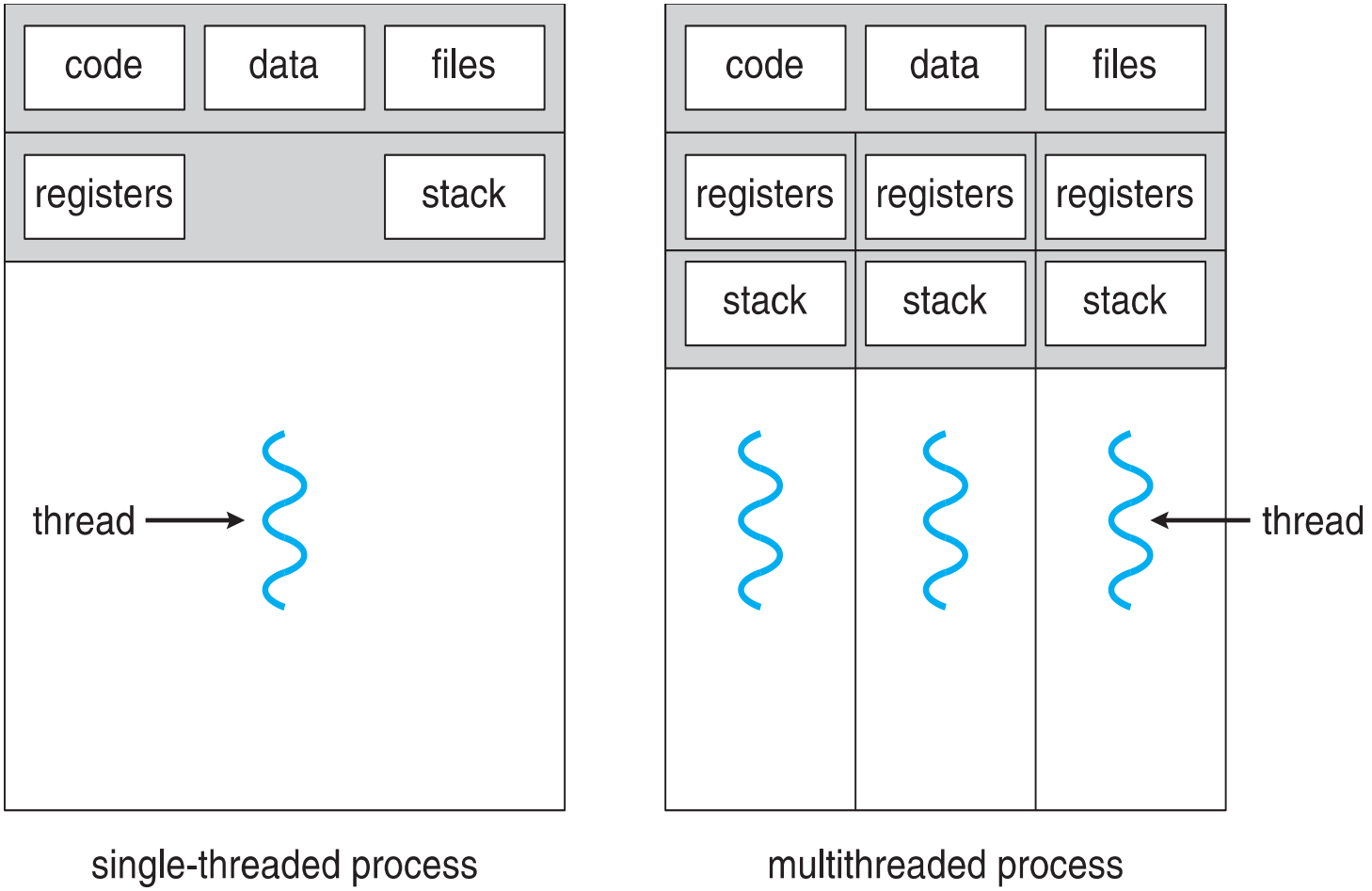


- **Parallelism on a multi-core system:**





Single and Multithreaded Processes



Amdahl's Law

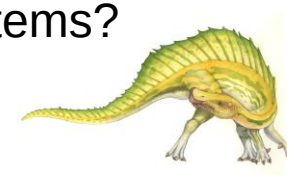
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?



User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

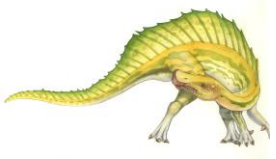


Illustration of user threads—Sybase ASE

- MFSPASE16.log:00:0010:00000:00000:2015/02/19 17:40:31.46 kernel I/O controller 2 (NetController) is running as task 1624634048 on thread 10 (LWP 20644033).
- MFSPASE16.log:00:0011:00000:00000:2015/02/19 17:40:31.47 kernel I/O controller 3 (DiskController) is running as task 1624635728 on thread 11 (LWP 88408255).
- MFSPASE16.log:00:0002:00000:00000:2015/02/19 17:40:31.83 kernel Thread 2 (LWP 76284109) of Threadpool syb_default_pool online as engine 0
- MFSPASE16.log:00:0004:00000:00000:2015/02/19 17:40:49.93 kernel Thread 4 (LWP 41549831) of Threadpool syb_default_pool online as engine 2
- MFSPASE16.log:00:0003:00000:00000:2015/02/19 17:40:49.95 kernel Thread 3 (LWP 36700339) of Threadpool syb_default_pool online as engine 1
- MFSPASE16.log:00:0005:00000:00000:2015/02/19 17:40:49.96 kernel Thread 5 (LWP 57344237) of Threadpool syb_default_pool online as engine 3

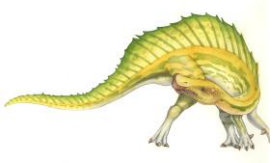
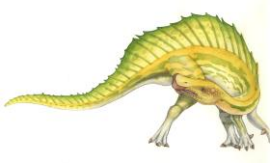


Illustration of user threads – Sybase ASE

- ps -emo THREAD
- sybase 13631516 14024954 - A 0 60 18 * 240001 pts/0 - /software/sybase/ASE-16_0/bin/dataserver -d/software/sybase/data/master.dat
- - - - 20644033 S 0 60 1 f1000e00039ac420 410400 - - - **net controller (IOCP)**
- - - - 33882361 S 0 60 1 f1000f0a10020540 8c10400 - - -
- - - - 34537587 S 0 60 1 - 418400 - - -
- - - - 36700339 S 0 60 1 f1000f0a10023040 8410400 - - - **engine 1**
- - - - 38862939 S 0 60 1 f1000f0a10025140 8430400 - - -
- - - - 41549831 S 0 60 1 f1000f0a10027a40 8410400 - - - **engine 2**
- - - - 41746551 S 0 60 1 f1000f0a10027d40 8410400 - - -
- - - - 44171397 S 0 60 1 f1000f0a1002a240 8410400 - - -
- - - - 47120455 S 0 60 1 f1000f0a1002cf40 8410400 - - -
- - - - 52297931 S 0 60 1 f1000f0a10031e40 8410400 - - -
- - - - 57344237 S 0 60 1 f1000f0a10036b40 8410400 - - - **engine 3**
- - - - 61407355 S 0 60 1 f1000f0a1003a940 8410400 - - -
- - - - 63307969 S 0 60 1 f1000a03e02ee5e8 410400 - - -
- - - - 75300877 S 0 60 1 f1000f0a10047d40 8410400 - - -
- - - - 76284109 S 0 60 1 f1000f0a10048c40 8410400 - - - **engine 0**
- - - - 79823021 S 0 60 1 f1000f0a1004c240 8410400 - - -
- - - - 80150685 S 0 60 1 - 418400 - - -
- - - - 88408255 S 0 60 1 f1000e00038d4420 410400 - - - **disk controller (IOCP)**

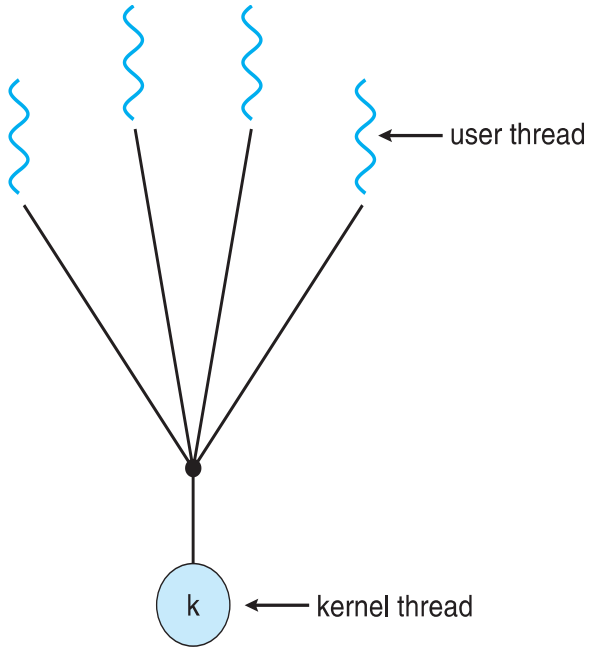
Multithreading Models

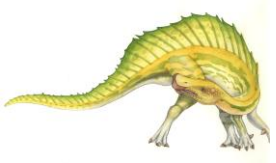
- Many-to-One
- One-to-One
- Many-to-Many



Many-to-One

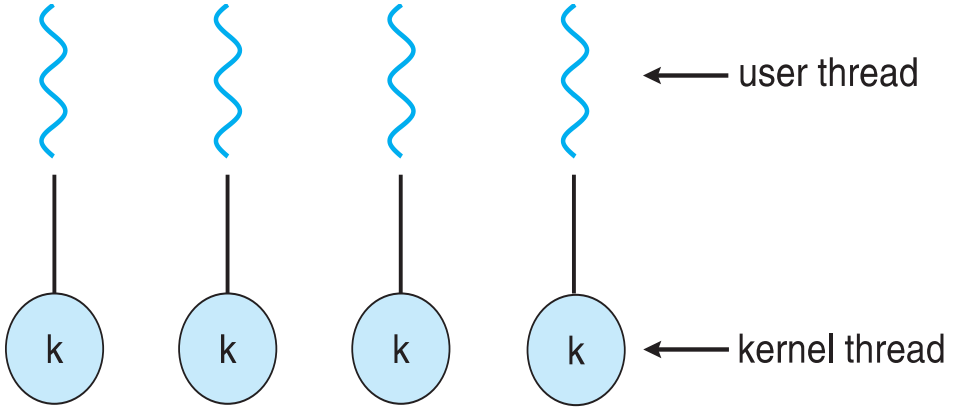
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

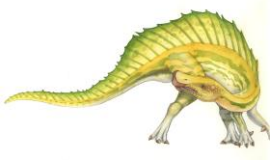




One-to-One

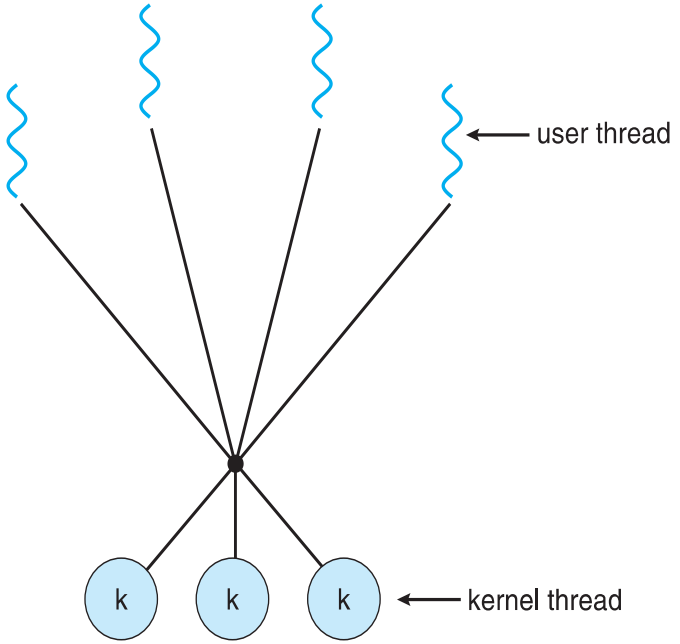
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

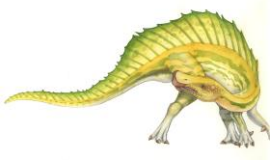




Many-to-Many Model

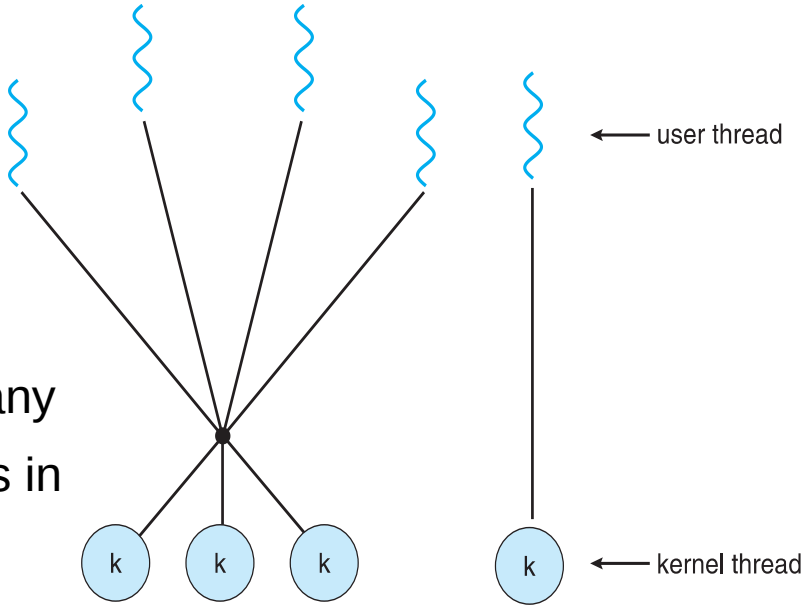
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

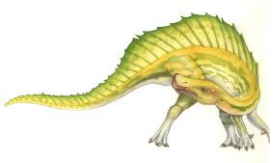




Two-level Model

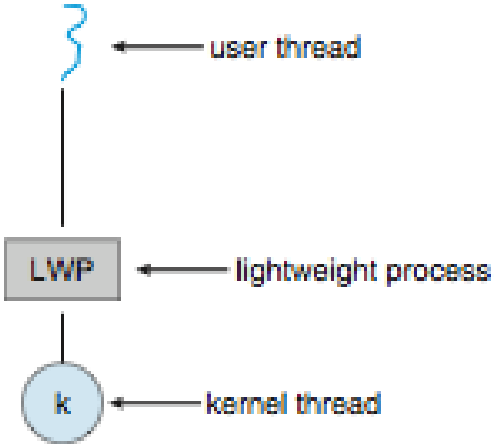
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier
- PB note: one:one and many:many are usually configurable options, as in Solaris, AIX, most Linux.

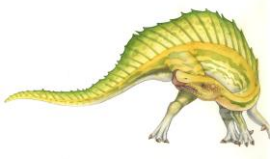




Scheduler Activations (moved up in deck)

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads





Thread scope in UNIX and Linux

LINUX, SOLARIS and AIX currently support "PROCESS (many-one) and SYSTEM (one-one) thread scope.

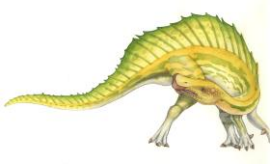
The pthread_attr_setscope() function sets the contention scope attribute of the thread attributes object referred to by attr to the value specified in scope. The contention scope attribute defines the set of threads against which a thread competes for resources such as the CPU. POSIX.1-2001 specifies two possible values for scope:

PTHREAD_SCOPE_SYSTEM

The thread competes for resources with all other threads in all processes on the system that are in the same scheduling allocation domain (a group of one or more processors). PTHREAD_SCOPE_SYSTEM threads are scheduled relative to one another according to their scheduling policy and priority.

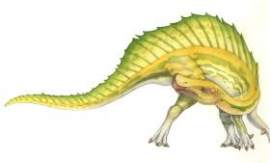
PTHREAD_SCOPE_PROCESS

The thread competes for resources with all other threads in the same process that were also created with the PTHREAD_SCOPE_PROCESS contention scope. PTHREAD_SCOPE_PROCESS threads are scheduled relative to other threads in the process according to their scheduling policy and priority.
POSIX.1-2001 leaves it unspecified how these threads contend with other threads in other process on the system or with other threads in the same process that were created with the PTHREAD_SCOPE_SYSTEM contention scope.



Thread Libraries

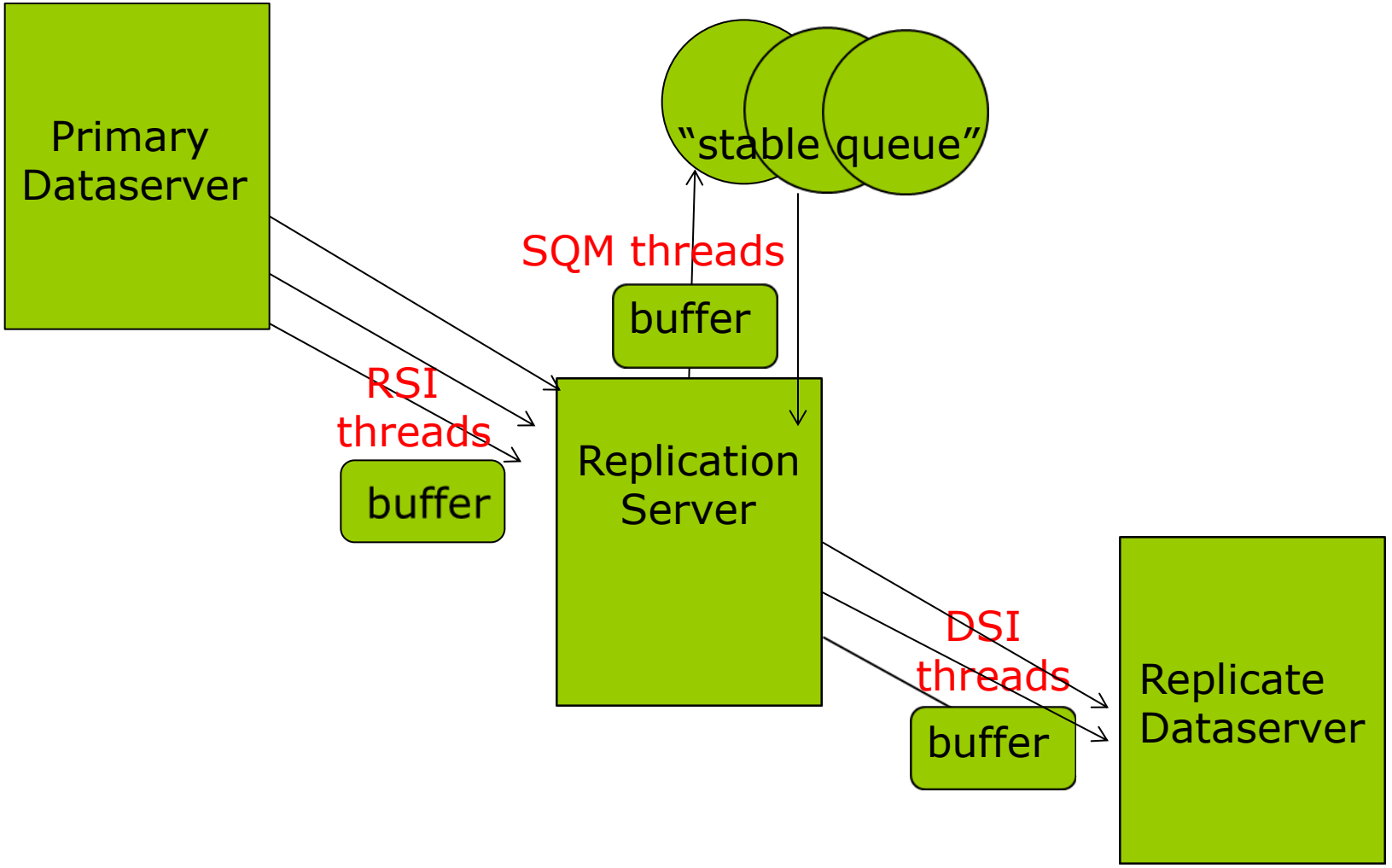
- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS



Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

A complex pthread application: SAP Replication Server

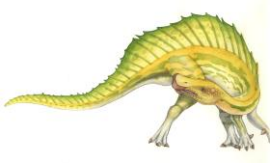


Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

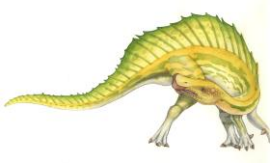


Modification to pthread example 4.25-4.26 to create and join multiple threads

```
#define NUM_THREADS 50
……..
int main(int argc, char *argv[])  /*in this modification pass it the number of threads =<50 */
{
pthread_t workers[NUM_THREADS]; /* the thread identifier */
pthread_attr_t attr; /* set of attributes for the thread */
…….
pthread_attr_init(&attr);
……….
/* create the threads */
for(i=0;i<atoi(argv[1]);i++)
pthread_create(&workers[i],&attr,runner,argv[1]);

/* now wait for the thread to exit */
for(i=0; i<atoi(argv[1]); i++)
pthread_join(workers[i],NULL);
```

Implicit Threading

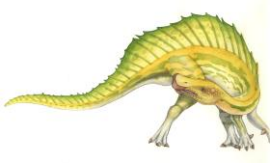
- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), **`java.util.concurrent`** package



Thread Pools

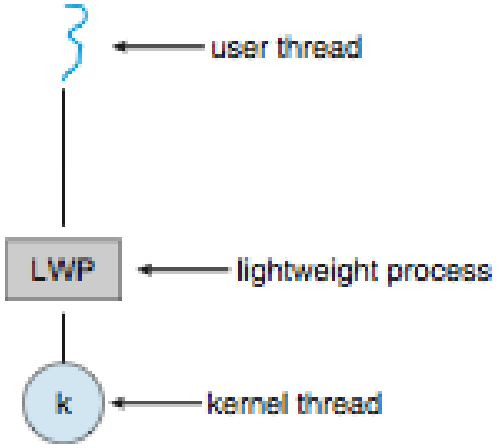
- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.Tasks could be scheduled to run periodically
- Windows API supports thread pools:

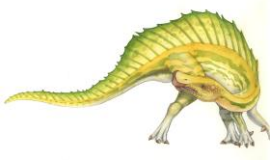
```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```



Scheduler Activations (moved up again)

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads



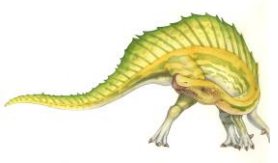


Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads



Pthread illustration from the Chapter 4 exercises showing fork+ thread creation

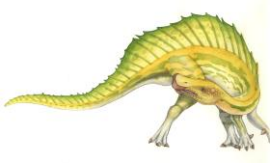
the application? Explain.

4.15 Consider the following code segment:

```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
  fork();
  thread_create( . . .);
}
fork();
```

a. How many unique processes are created?

b. How many unique threads are created?



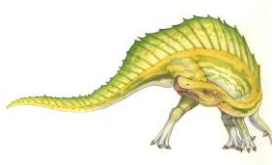
Another illustration of fork and thread – what are the outputs of Line C and Line P?

```
#include <pthread.h>
#include <stdio.h>
#include <types.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

Figure 4.16 C program for Exercise 4.17.

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
  4. Or the signal can be masked to be ignored.
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process
  - Remember – the wait() system call is a kind of “canned” signal handler for the child processes

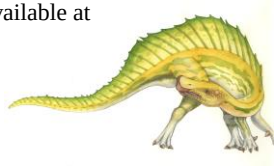


Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
- How some of these topics are handled in posix:
  - Synchronous signals are delivered to the affected thread
    - Errors
    - Internal pthread_kill signals (deprecated)
  - Asynchronous signals coming from “outside”
    - Delivered to the process as a whole
    - Handled by each thread depending on setting of its is signal mask.
      - Ignorer
      - Default
      - User defined
      - Blocked until ready to handle
- Important limitation of posix thread signal handling –no IPC functions are defined

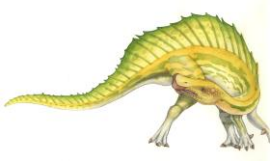
"kill" system call sends signals to processes and threads

Signal	Code on Linux	Default Action	Description
SIGABRT	6	Terminate (core dump)	Process abort signal
SIGALRM	14	Terminate	Alarm clock
SIGBUS	10	Terminate (core dump)	Access to an undefined portion of a memory object
SIGCHLD	18	Ignore [9]	Child process terminated, stopped,
SIGCONT	25	Continue	Continue executing, if stopped.
SIGFPE	8	Terminate (core dump)	Erroneous arithmetic operation.
SIGHUP	1	Terminate	Hangup.
SIGILL	4	Terminate (core dump)	Illegal instruction.
SIGINT	2	Terminate	Terminal interrupt signal.
SIGKILL	9	Terminate	Kill (cannot be caught or ignored).
SIGPIPE	13	Terminate	Write on a pipe with no one to read it.
SIGQUIT	3	Terminate (core dump)	Terminal quit signal.
SIGSEGV	11	Terminate (core dump)	Invalid memory reference.
SIGSTOP	23	Stop	Stop executing (cannot be caught or ignored).
SIGTERM	15	Terminate	Termination signal.
SIGTSTP	20	Stop	Terminal stop signal.
SIGTTIN	26	Stop	Background process attempting read.
SIGTTOU	27	Stop	Background process attempting write.
SIGUSR1	16	Terminate	User-defined signal 1.
SIGUSR2	17	Terminate	User-defined signal 2.
SIGPOLL	22	Terminate	Pollable event.
SIGPROF	29	Terminate	Profiling timer expired.
SIGSYS	12	Terminate (core dump)	Bad system call.
SIGTRAP	5	Terminate (core dump)	Trace/breakpoint trap.
SIGURG	21	Ignore	High bandwidth data is available at a socket.
SIGVTALRM	28	Terminate	Virtual timer expired.
SIGXCPU	30	Terminate (core dump)	CPU time limit exceeded.
SIGXFSZ	31	Terminate (core dump)	File size limit exceeded



The “kill” command’s numeric arguments control which processes or threads get the signal

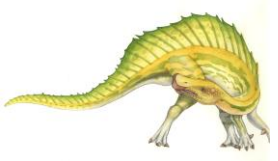
- The command **kill** sends the specified *signal* to the specified processes or process groups. If no signal is specified, the TERM signal is sent. This TERM signal will kill processes that do not catch it; for other processes it may be necessary to use the KILL signal (number 9), since this signal cannot be caught. Most modern shells have a builtin kill function, with a usage rather similar to that of the command described here. The **--all**, **--pid**, and **--queue** options, and the possibility to specify processes by command name, are local extensions. If *signal* is 0, then no actual signal is sent, but error checking is still performed.
- The list of processes to be signaled can be a mixture of names and pids.
- *pid* Each *pid* can be one of four things: *n* where *n* is larger than 0. The process with pid *n* is signaled. **0** All processes in the current process group are signaled. **-1** All processes with a pid larger than 1 are signaled. **-***n* where *n* is larger than 1. All processes in process group *n* are signaled. When an argument of the form '-n' is given, and it is meant to denote a process group, either a signal must be specified first, or the argument must be preceded by a '--' option, otherwise it will be taken as the signal to send.
- *name* All processes invoked using this *name* will be signaled.
- It is not possible to send a signal to an explicitly selected thread in a multithreaded process using the kill(2) syscall. If kill(2) is used to send a signal to a thread group, then the kernel selects an arbitrary member of the thread group that has not blocked the signal.
- The command kill(1) as well as syscall kill(2) accept a TID (thread ID, see gettid(2)) as an argument. In this case the kill behavior is not changed and the signal is also delivered to the thread group rather than to the specified thread.



Thread Cancellation

- Terminating a thread before it has finished
- Preferable to using “pthread_kill()
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread cc

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```



Thread Cancellation (Cont.)

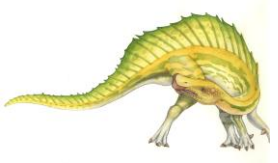
- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **`static`** data
  - TLS is unique to each thread



Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **`struct task_struct`** points to process data structures (shared or unique)