Exercise (could be a quiz)

In the code below, assume that *(i)* all `fork` and `execvp` statements execute successfully, *(ii)* the program arguments of `execvp` do not spawn more processes or print out more characters, and *(iii)* all `pid` variables are initialized to `0`.

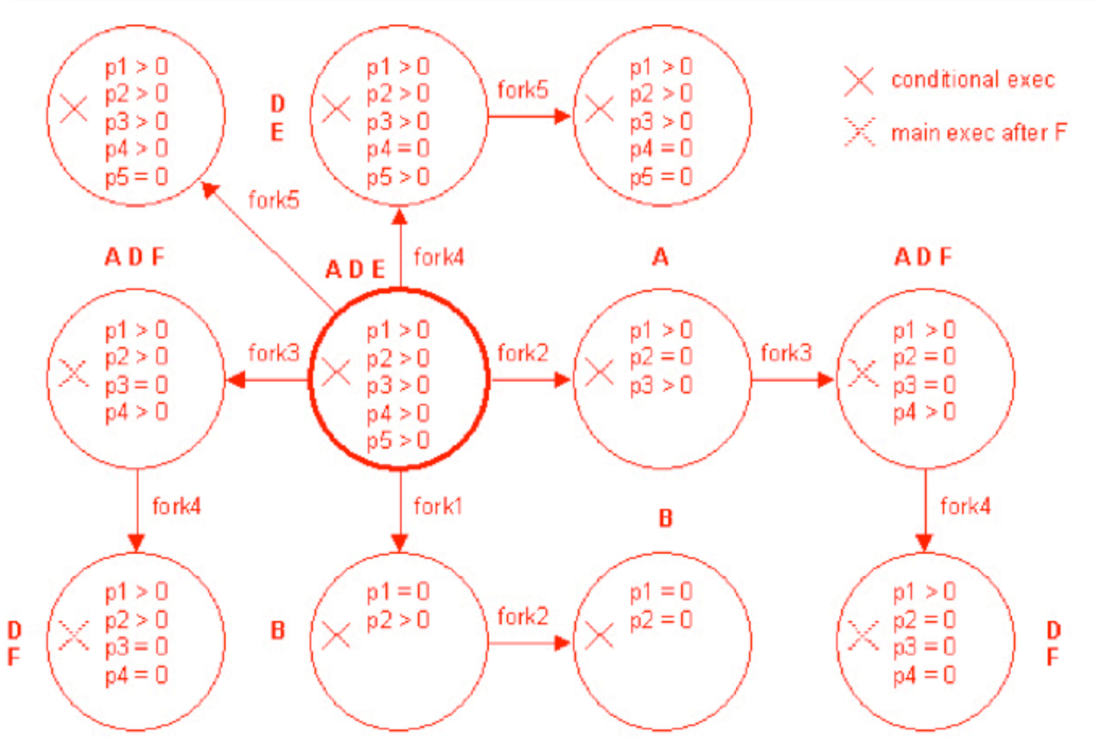
a. What is the total number of processes that will be created by the execution of this code?
b. How many of each character 'A' to 'G' will be printed out?

```
void main()
{
      ...
      pid1 = fork();
      pid2 = fork();
      if (pid1 != 0) {
          pid3 = fork();
          printf("A\n");
      } else {
          printf("B\n");
          execvp(...);
      }
      if (pid2 == 0 && pid3 != 0) {
          execvp(...);
          printf("C\n");
      }
      pid4 = fork();
      printf("D\n");
      if (pid3 != 0) {
          printf("E\n");
          pid5 = fork();
          execvp(...);
      }
      printf("F\n");
      execvp(...);
      pid6 = fork();
      printf("G\n");
      if (pid6 == 0)
          pid7 = fork();
}
```



```
void main()
{
      ...
      pid1 = fork();
      pid2 = fork();
      if (pid1 != 0) {
          pid3 = fork();
          printf("A\n");
      } else {
          printf("B\n");
          execvp(...);
      }
      if (pid2 == 0 && pid3 != 0) {
          execvp(...);
          printf("C\n");
      }
      pid4 = fork();
      printf("D\n");
      if (pid3 != 0) {
          printf("E\n");
          pid5 = fork();
          execvp(...);
      }
      printf("F\n");
      execvp(...);
      pid6 = fork();
      printf("G\n");
      if (pid6 == 0)
          pid7 = fork();
}
```

Solution



CSE 421/521 - Operating Systems
Fall 2013

LECTURE - IV

THREADS

Tevfik Koşar

University at Buffalo

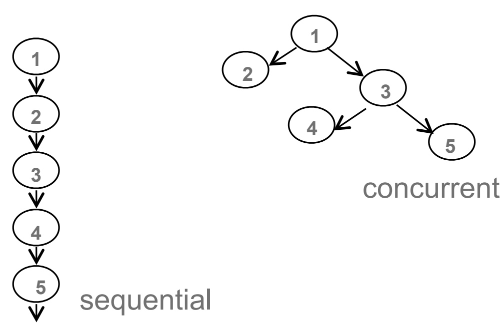
September 12th, 2013

Roadmap

- Threads
  - Why do we need them?
  - Threads vs Processes
  - Threading Examples
  - Threading Implementation & Multi-threading Models
  - Other Threading Issues
    - Thread cancellation
    - Signal handling
    - Thread pools
    - Thread specific data

Concurrent Programming

- In certain cases, a single application may need to run several tasks at the same time



Motivation

- Increase the performance by running more than one tasks at a time.
  - divide the program to n smaller pieces, and run it n times faster using n processors
- To cope with independent physical devices.
  - do not wait for a blocked device, perform other operations at the background

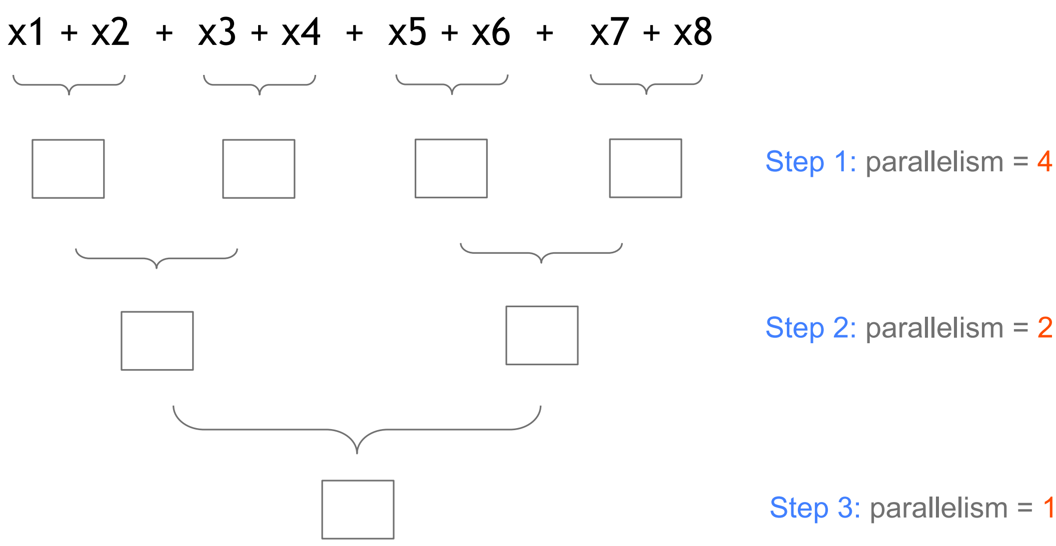
Divide and Compute

x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8

How many operations with sequential programming?
7

Step 1: x1 + x2
Step 2: x1 + x2 + x3
Step 3: x1 + x2 + x3 + x4
Step 4: x1 + x2 + x3 + x4 + x5
Step 5: x1 + x2 + x3 + x4 + x5 + x6
Step 6: x1 + x2 + x3 + x4 + x5 + x6 + x7
Step 7: x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8

Divide and Compute



Gain from parallelism

In theory:

- dividing a program into n smaller parts and running on n processors results in n time speedup

In practice:

- This is not true, due to
  - Communication costs
  - Dependencies between different program parts
    - Eg. the addition example can run only in log(n) time not 1/n

Concurrent Programming

- Implementation of concurrent tasks:
  - as separate programs
  - as a set of processes or threads created by a single program
- Execution of concurrent tasks:
  - on a single processor using multiple threads
  - ➔ Multithreaded programming
  - on several processors in close proximity
  - ➔ Parallel computing
  - on several processors distributed across a network
  - ➔ Distributed computing

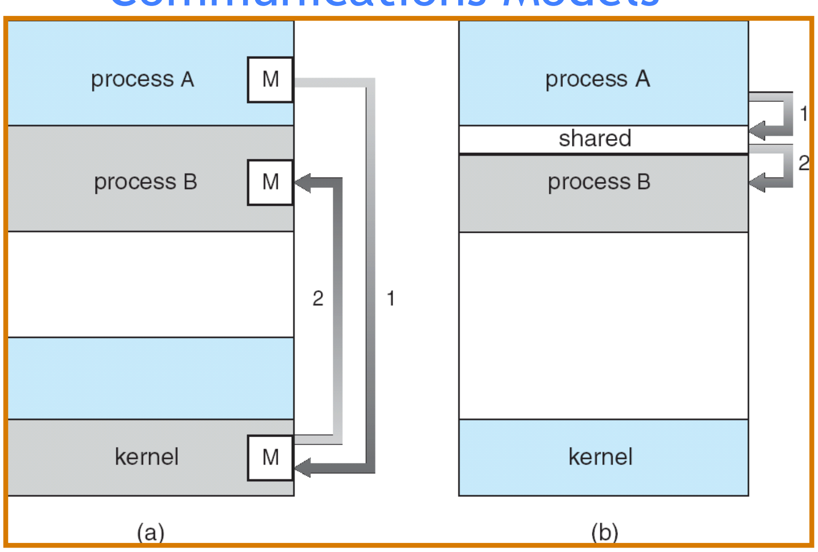
Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Disadvantage
  - Synchronization issues and race conditions

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Shared Memory: by using the same address space and shared variables
- Message Passing: processes communicate with each other without resorting to shared variables

Communications Models



a) Message Passing b) Shared Memory

Message Passing

- Message Passing facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Two types of Message Passing
  - direct communication
  - indirect communication

Message Passing – direct communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional
- Symmetrical vs Asymmetrical direct communication
  - **send** (*P, message*) – send a message to process P
  - **receive**(id, *message*) – receive a message from any process
- Disadvantage of both: limited modularity, hardcoded

Message Passing - indirect communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Indirect Communication *(cont.)*

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

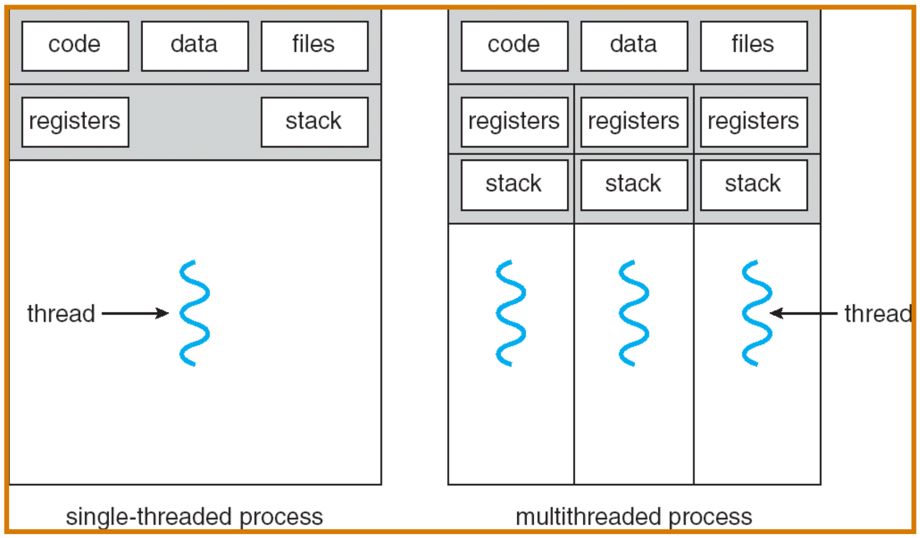
Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Concurrency with Threads

- In certain cases, a single application may need to run several tasks at the same time
  - Creating a new process for each task is time consuming
  - Use a single process with multiple threads
    - faster
    - less overhead for creation, switching, and termination
    - share the same address space

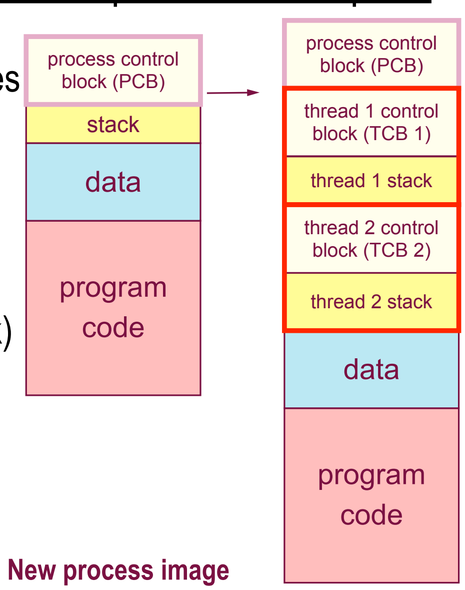
Single and Multithreaded Processes



New Process Description Model

➢ <u>Multithreading requires changes in the process description model</u>

- ✓ each thread of execution receives its own control block and stack
  - own execution state ("Running", "Blocked", etc.)
  - own copy of CPU registers
  - own execution history (stack)
- ✓ the process keeps a global control block listing resources currently used



New process image

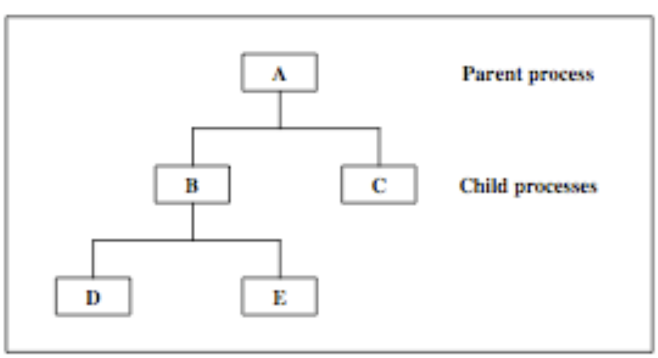
Per-process vs per-thread items

- Per-process items and per-thread items in the control block structures
  - ✓ process identification data + thread identifiers
    - numeric identifiers of the process, the parent process, the user, etc.
  - ✓ CPU state information
    - user-visible, control & status registers
    - stack pointers
  - ✓ process control information
    - scheduling: state, priority, awaited event
    - used memory and I/O, opened files, etc.
    - pointer to next PCB

Multi-process model

Process Spawning:

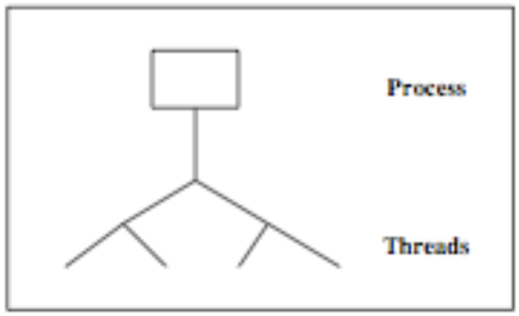
Process creation involves the following four main actions:
- **setting up the process control block,**
- **allocation of an address space and**
- **loading the program into the allocated address space and**
- **passing on the process control block to the scheduler**



Multi-thread model

Thread Spawning:

- **Threads are created *within and belonging to* processes**
- **All the threads created within one process share the resources of the process including the address space**
- **Scheduling is performed on a per-thread basis.**
- **The thread model is a *finer grain scheduling model* than the process model**
- **Threads have a similar *lifecycle* as the processes and will be managed mainly in the same way as processes are**



Threads vs Processes

- A common terminology:
  - Heavyweight Process = Process
  - **Lightweight Process = Thread**

Advantages (Thread vs. Process):

- Much quicker to create a thread than a process
  - spawning a new thread only involves allocating a new stack and a new CPU state block
- Much quicker to switch between threads than to switch between processes
- Threads share data easily

Disadvantages (Thread vs. Process):

- Processes are more flexible
  - They don't have to run on the same processor
- No security between threads: One thread can stomp on another thread's data
- For threads which are supported by user thread package instead of the kernel:
  - If one thread blocks, all threads in task block.

Thread Creation

- **pthread_create**

```
// creates a new thread executing start_routine
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

- **pthread_join**

```
// suspends execution of the calling thread until the target
// thread terminates
int pthread_join(pthread_t thread, void **value_ptr);
```


Thread Example

```
    int main()
{
    pthread_t thread1, thread2;  /* thread variables */


    pthread_create (&thread1, NULL, (void *) &print_message_function,
                                                  (void*)"hello ");
    pthread_create (&thread2, NULL, (void *) &print_message_function,
                                                  (void*)"world!\n");

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    exit(0);
}
```

Why use pthread_join?
To force main block to wait for both threads to terminate, before it exits. If main block exits, both threads exit, even if the threads have not finished their work.

Exercise

Consider a process with two concurrent threads T1 and T2. The code being executed by T1 and T2 is as follows:

Shared Data:
X:= 5; Y:=10;

T1:	T2:
Y = X+1;	U = Y-1;
X = Y;	Y = U;
Write X;	Write Y;

Assume that each assignment statement on its own is executed as an atomic operation. What is the outputs of this process?

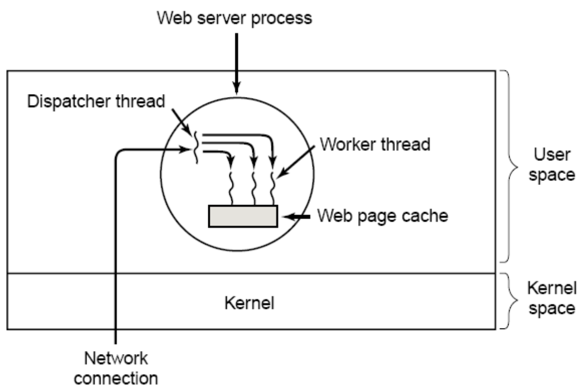
Solution

All six statements can be executed in any order. Possible outputs are:

1) 65
2) 56
3) 55
4) 99
5) 66
6) 69
7) 96

Threading Examples

- **Web server**
  - ✓ as each new request comes in, a "dispatcher thread" spawns a new "worker thread" to read the requested file (worker threads may be discarded or recycled in a "thread pool")

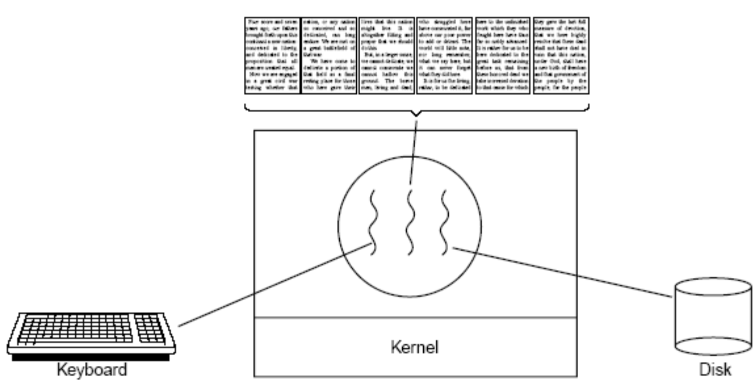


Tanenbaum, A. S. (2001) *Modern Operating Systems (2nd Edition).*

A multithreaded Web server

Threading Examples

- **Word processor**
  - ✓ one thread listens continuously to keyboard and mouse events to refresh the GUI; a second thread reformats the document (to prepare page 600); a third thread writes to disk periodically

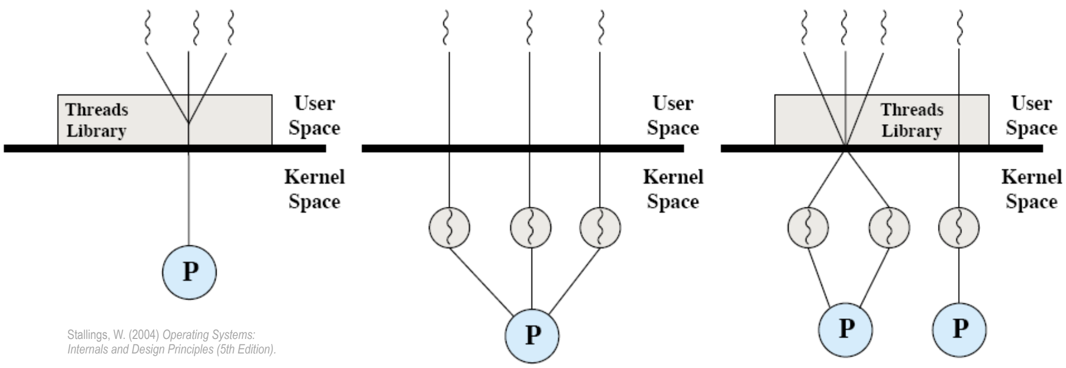


Tanenbaum, A. S. (2001) *Modern Operating Systems (2nd Edition).*

A word processor with three threads

Thread Implementation

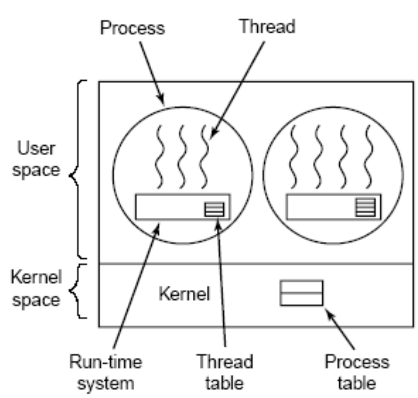
- **Two broad categories of thread implementation**
  - User-Level Threads (ULTs)
  - Kernel-Level Threads (KLTs)



Pure user-level (ULT), pure kernel-level (KLT) and combined-level (ULT/KLT) threads

Thread Implementation

- **User-Level Threads (ULTs)**
  - the kernel is not aware of the existence of threads, it knows only processes with one thread of execution (one PC)
  - each user process manages its own private thread table
  - light thread switching: does not need kernel mode privileges
  - cross-platform: ULTs can run on any underlying O/S
  - if a thread blocks, the entire process is blocked, including all other threads in it

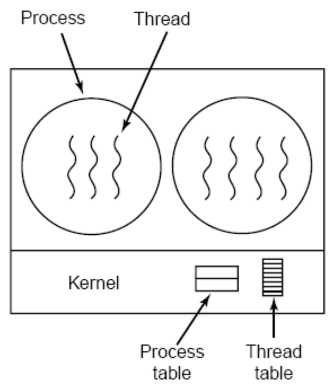


Tanenbaum, A. S. (2001) Modern Operating Systems (2nd Edition).

A user-level thread package

Thread Implementation

- <u>Kernel-Level Threads</u>
  - ✓ the kernel knows about and manages the threads: creating and destroying threads are system calls
  - <u>fine-grain scheduling</u>, done on a thread basis
  - <u>if a thread blocks, another one can be scheduled</u> without blocking the whole process
  - <u>heavy thread switching</u> involving mode switch



A kernel-level thread package

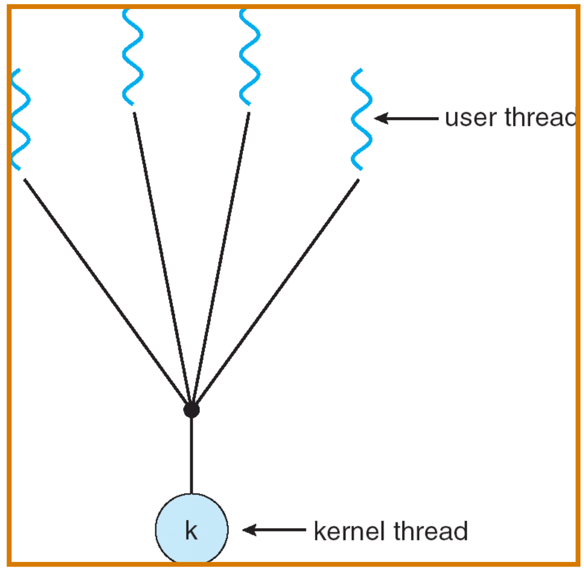
Tanenbaum, A. S. (2001) *Modern Operating Systems (2nd Edition).*

Different Multi-threading Models

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

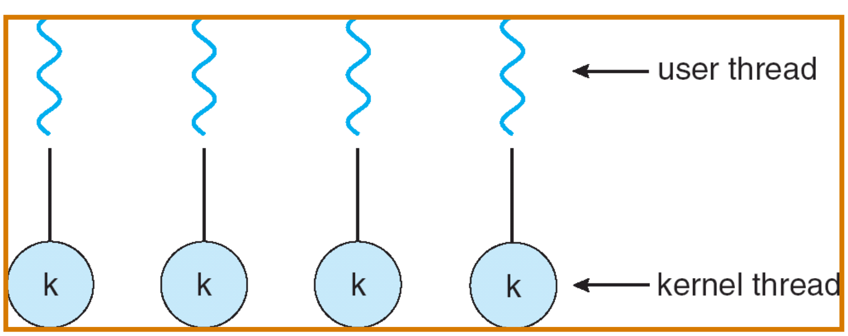
Many-to-One Model

- Several user-level threads mapped to single kernel thread
- Thread management in user space → efficient
- If a thread blocks, entire process blocks
- One thread can access the kernel at a time → limits parallelism
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



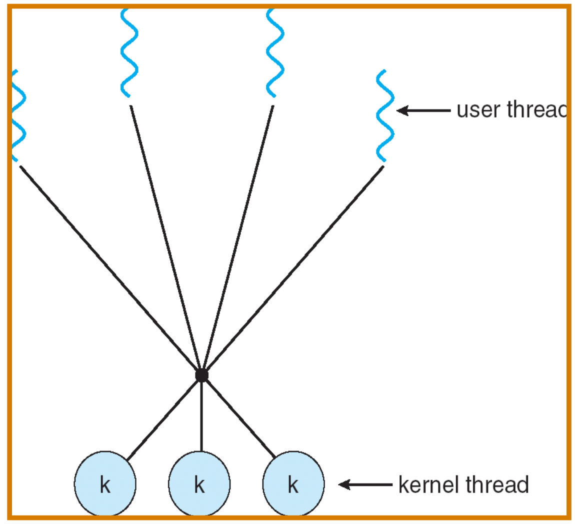
One-to-One Model

- Each user-level thread maps to a kernel thread
- A blocking thread does not block other threads
- Multiple threads can access kernel concurrently → increased parallelism
- Drawback: Creating a user level thread requires creating a kernel level thread → increased overhead and limited number of threads
- Examples: Windows NT/XP/2000, Linux, Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to a smaller number of kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Increased parallelism as well as efficiency
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Threading Issues

- Thread pools
- Thread specific data
- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling

Thread Pools

- Threads come with some overhead as well
- Unlimited threads can exhaust system resources, such as CPU or memory
- Create a number of threads at process startup) and put them in a pool, where they await work
- When a server receives a request, it awakens a thread from this pool
- Advantages:
  - Usually 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
- Number of threads in the pool can be setup according to:
  - Number of CPUs, memory, expected number of concurrent requests

Semantics of fork() and exec()

- Semantics of **fork()** and **exec()** system calls change in a multithreaded program
  - Eg. if one thread in a multithreaded program calls fork()
    - Should the new process duplicate all threads?
    - Or should it be single-threaded?
  - Some UNIX systems implement two versions of fork()
  - If a thread executes exec() system call
    - Entire process will be replaced, including all threads

Thread Cancellation

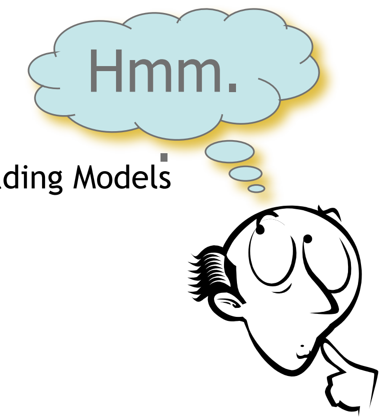
- Terminating a thread before it has finished
  - If one thread finishes searching a database, others may be terminated
  - If user presses a button on a web browser, web page can be stopped from loading further
- Two approaches to cancel the target thread
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - More controlled and safe

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- All signals follow this pattern:
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Once delivered, a signal must be handled
- In multithreaded systems, there are 4 options:
  - 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

Summary

- Why do we need them?
- Threads vs Processes
- Threading Examples
- Threading Implementation & Multi-threading Models
- Other Threading Issues
  - Thread cancellation
  - Signal handling
  - Thread pools
  - Thread specific data



- HW1 out today
- Next Lecture: CPU Scheduling
- Reading Assignment: Chapter 5 from Silberschatz.

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR