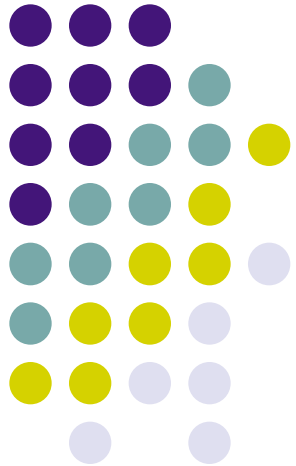
Last topic for processors

Threads



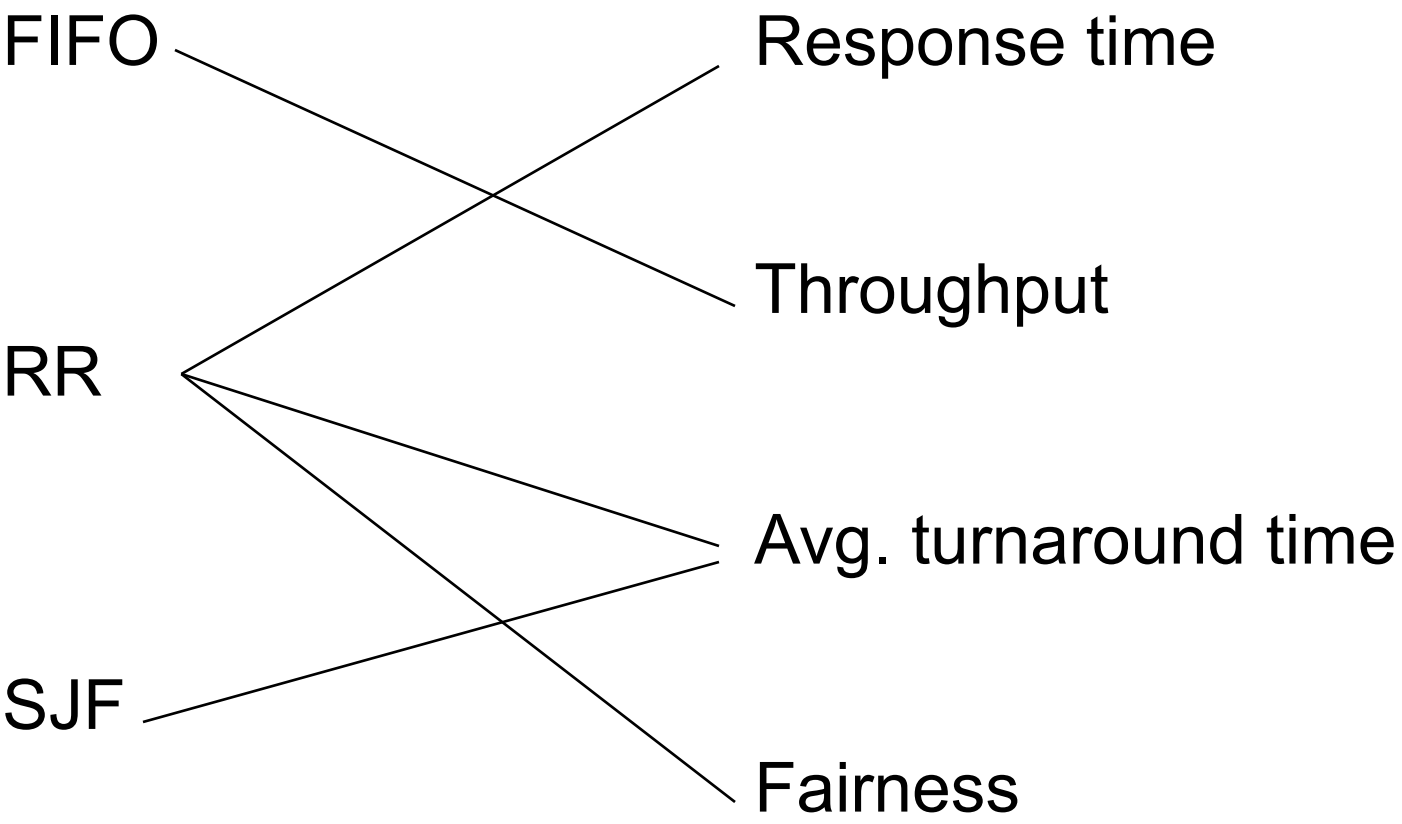
ECE469, Feb 14

Yiying Zhang

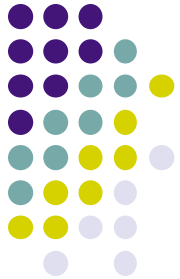
17. threads control block

33. Three multithreading models

[lec10] Scheduling policies

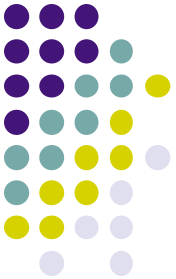


[lec10] Priority Scheduling



- To accommodate the spirits of SJF/RR/FIFO
- The method
  - Assign each process a *priority*
  - Run the process with highest priority in ready queue first
    - Use FIFO for processes with equal priority
  - Adjust priority dynamically
    - To deal with *all* issues: e.g. aging, I/O wait raises priority
- Advantage
  - Flexibility: Not all processes are “born” equal
- Challenge?

[lec10] Approach 2: Multilevel Feedback Queue (MLFQ)



- Problem: how to change priority?

- Jobs start at highest priority queue
- Feedback
  - If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
  - If a job gives up the CPU before the time slice is up, it stays at the same priority level.
  - After a long time period, move all the jobs in the system to the topmost queue (aging)

Outline

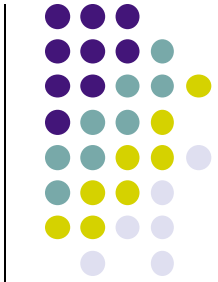
- Why threads?
- What are threads?
- Programming with threads
- Pthread code example

- Thread implementation
- Multithreading models
- Context switch threads

Read Assignment

- Dinosaur Chapter 4

- Comet Chapters 26, 27

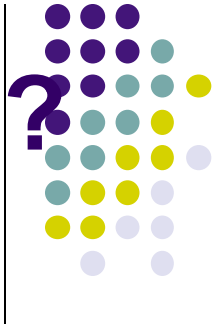


Processes

- A process contains everything needed for execution
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC (program counter), SP (stack pointer), regs, etc.)
  - Each of these resources is exclusive to the process

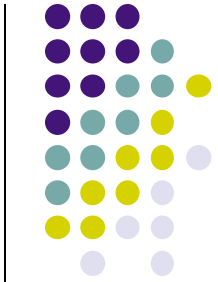
- Yet sometimes processes may wish to cooperate
  - But how to communicate? Each process is an island
  - The OS needs to intervene to bridge the gap
  - OS provides system calls to support Inter-Process Communication (IPC)

How do processes communicate?



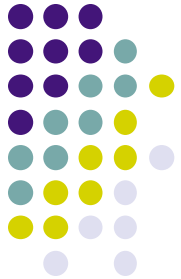
- At process creation time
  - Parents get one chance to pass everything at fork()
- OS provides generic mechanisms to communicate
  - Shared Memory: multiple processes can read/write same physical portion of memory; implicit channel
    - System call to declare shared region
    - No OS mediation required once memory is mapped
  - Message Passing: explicit communication channel provided through send()/receive() system calls
    - A system call is required
- IPC is, in general, expensive due to the need for system calls
  - Although many OSes have various forms of lightweight IPC

Web server example



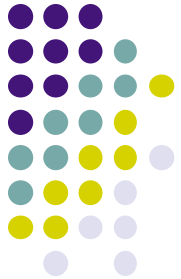
- How does a web server handle 1 request?
- A web server needs to handle many concurrent requests
- Solution 1:
  - Have the parent process fork as many processes as needed
  - Processes communicate with each other via inter-process communication

Parallel Programs



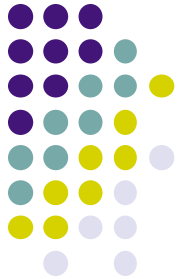
- To execute parallel programs we need to
  - Create several processes that execute in parallel
  - Use shared memory or message passing to share data
    - Notice: they are all part of the same computation!
  - Have the OS schedule these processes in parallel (logically or physically)
- This situation is very inefficient
  - Space: PCB, page tables, etc.
  - Time: create data structures, system calls for IPC, etc.

The Soul of a Process



- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)
- What don't they share?
  - Each has its own execution state: PC, SP, and registers
- Key idea: Why don't we separate the concept of a process from its execution state?
  - Process: address space, privileges, resources, etc.
  - Execution state: PC, SP, registers
- Exec state also called thread of control, or thread

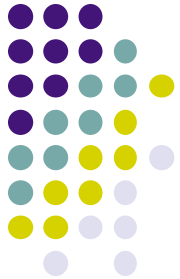
Threads



- Separate the concepts of a "thread of control" (PC, SP, registers) from the rest of the process (address space, resources, accounting, etc.)

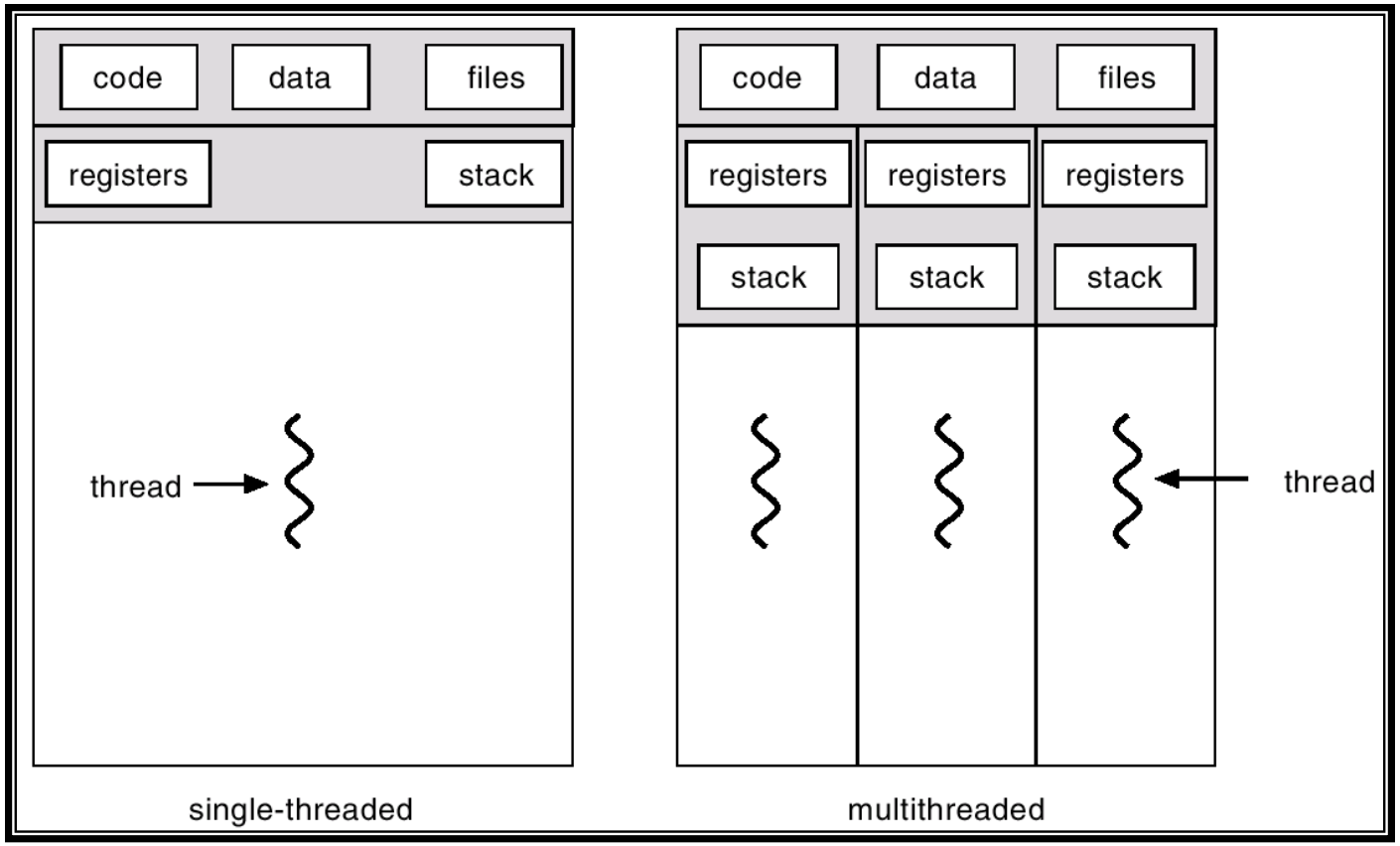
- Modern OSes support two entities:
  - the *task* (process), which defines an address space, a resource container, accounting info
  - the *thread* (lightweight process), which defines a single sequential execution stream within a task (process)

Threads vs. Processes

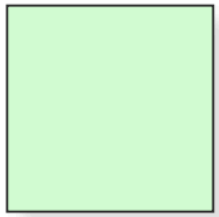


- There can be several threads in a single address space
- Threads are the unit of scheduling; tasks are containers (address space, other shared resources) in which threads execute
- In this model, a conventional (single-thread) process consists of a task and a single thread of control

Single and Multithreaded Processes



Thread Design Space

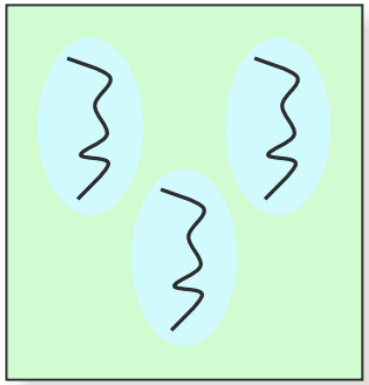


Address Space

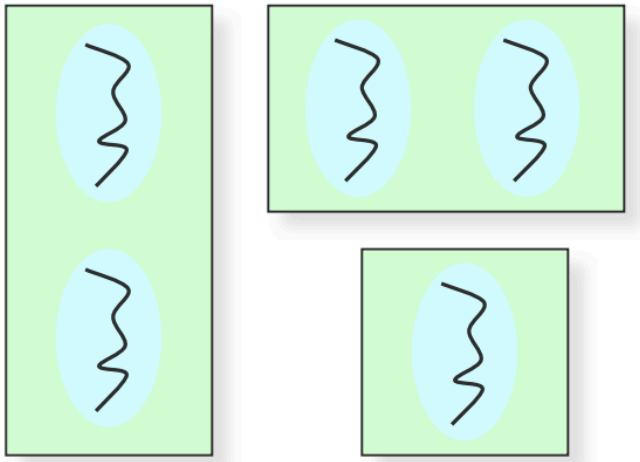
Thread

One Thread per Process
One Address Space
(MSDOS)

One Thread per Process
Many Address Spaces
(Early Unix)

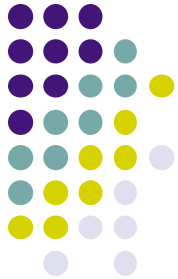


Many Threads per Process
One Address Space
(Java VM)



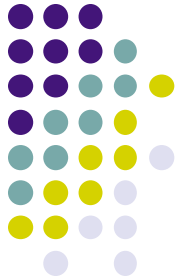
Many Threads per Process
Many Address Spaces
(Solaris, Linux, NT, MacOS)

[lec2] Process Control Block



- Process management info
  - State (ready, running, blocked)
  - PC & Registers
  - CPU scheduling info (priorities, etc.)
  - Parent info
- Memory management info
  - Segments, page table, stats, etc
  - Code, data, heap, execution stack
- I/O and file management
  - Communication ports, directories, file descriptors, etc.

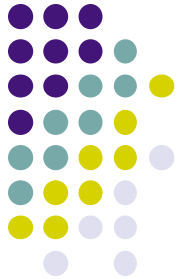
Thread Control Block



- Shared information
  - Process info: parent process
  - Memory: code/data segments, page table, and stats
  - I/O and file: comm ports, open file descriptors

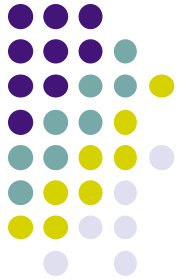
- Private state
  - State (ready, running and blocked)
  - PC, Registers
  - Execution stack

Break



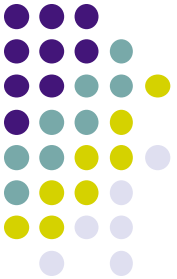
- How many trailing zeroes are there in 100! (100 factorial) ?

Programming with threads



- Flexible, but error-prone, since there no protection between threads
  - In C/C++,
    - automatic variables are private to each thread
    - global variables and dynamically allocated memory (malloc) are shared
- Need synchronization!

Pthread example



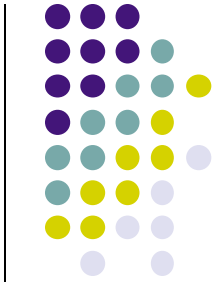
```
#include <pthread.h>

void *print_msg( void *ptr )
{ char *message; message = (char *) ptr; printf("%s \n", message); }

main() {
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int iret1, iret2;

    /* Create independent threads each of which will execute function */
    iret1 = pthread_create( &thread1, NULL, print_msg, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_msg, (void*) message2);

    /* Wait till threads are complete before main continues */
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2); exit(0);
}
```

Outline

- Why threads?
- What are threads?
- Programming with threads
- Pthread code example

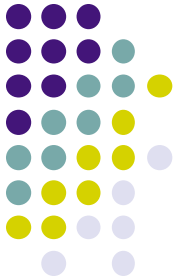
- Thread implementation
- 3 multithread models
- Context switch threads

Scheduling Threads



- No longer just scheduling processes, but threads
  - Kernel scheduler used to pick among PCBs
  - Now what?
- We have basically two options
  - Kernel explicitly selects among threads in a process
  - Hide threads from the kernel, and have a user-level scheduler inside each multi-threaded process
- Why do we care?
  - Think about the overhead of switching between threads
  - Who decides which thread in a process should go first?
  - What about blocking system calls?

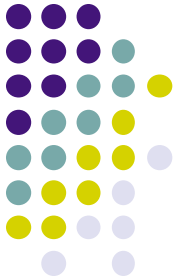
An Analogy: Family Car Rental



- Scenario (a day is 9am-5pm)
  - Avis rents a car to 2 family, Round-robin daily
  - Each family has 4 members, round-robin every 2 hrs

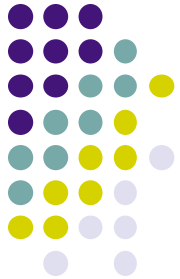
- Two ways of doing it:
  - Global scheduler: Avis schedules family for each day, family schedules among its members
    - Pros: efficient,
    - Cons: if a member has accidient at 9am?
  - Local scheduler: Avis schedules among 8 members

Thread Implementations



- User-level thread implementation
- Kernel-level thread implementation

User-Level Thread Implementation

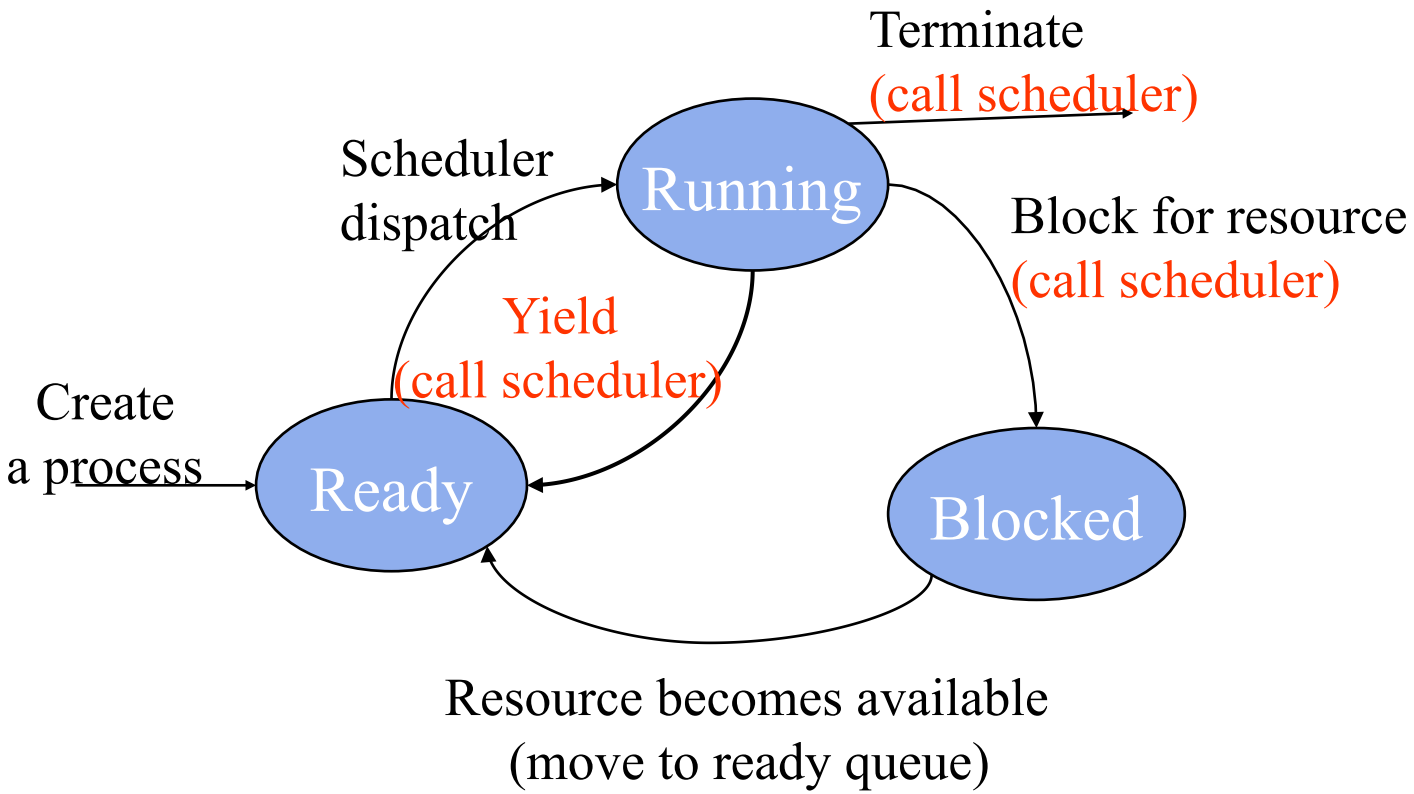


- User-level threads are managed entirely by a run-time system (a.k.a. user-level thread library)
  - Creation / scheduling
  - No kernel intervention (kernel sees single entity)

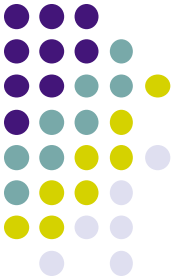
- Invisible to kernel
  - A thread represented inside process by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching, and synchronizing threads are done via user-level procedure call
  - User-level thread operations 100x faster than kernel thread

The big picture

- The kernel only sees one scheduling entity (which has many user threads inside)



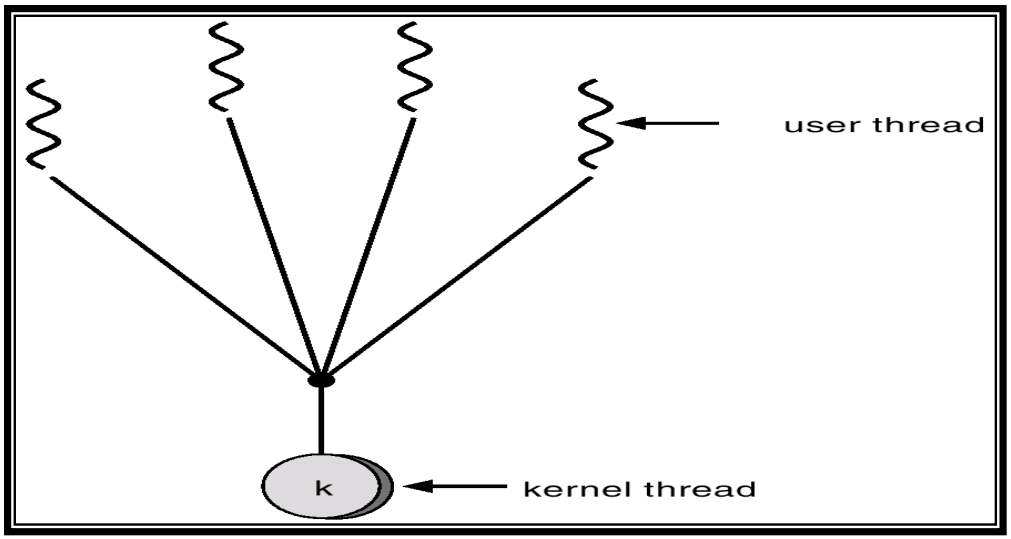
User-Level Threads Mapping to Kernel Threads



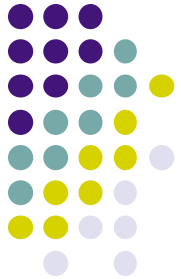
- Definition: Kernel thread is the kernel scheduling unit

- In user thread implementation, all user threads of the same process are effectively mapped to one kernel thread

- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris 2 *UI-threads*

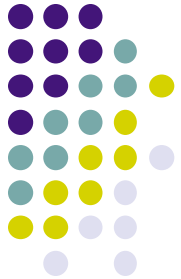


User-Level Thread Limitations



- What happens if a thread invokes a syscall?
  - A blocking syscall blocks the whole process!
- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  - Unscheduling a process with a thread holding a lock

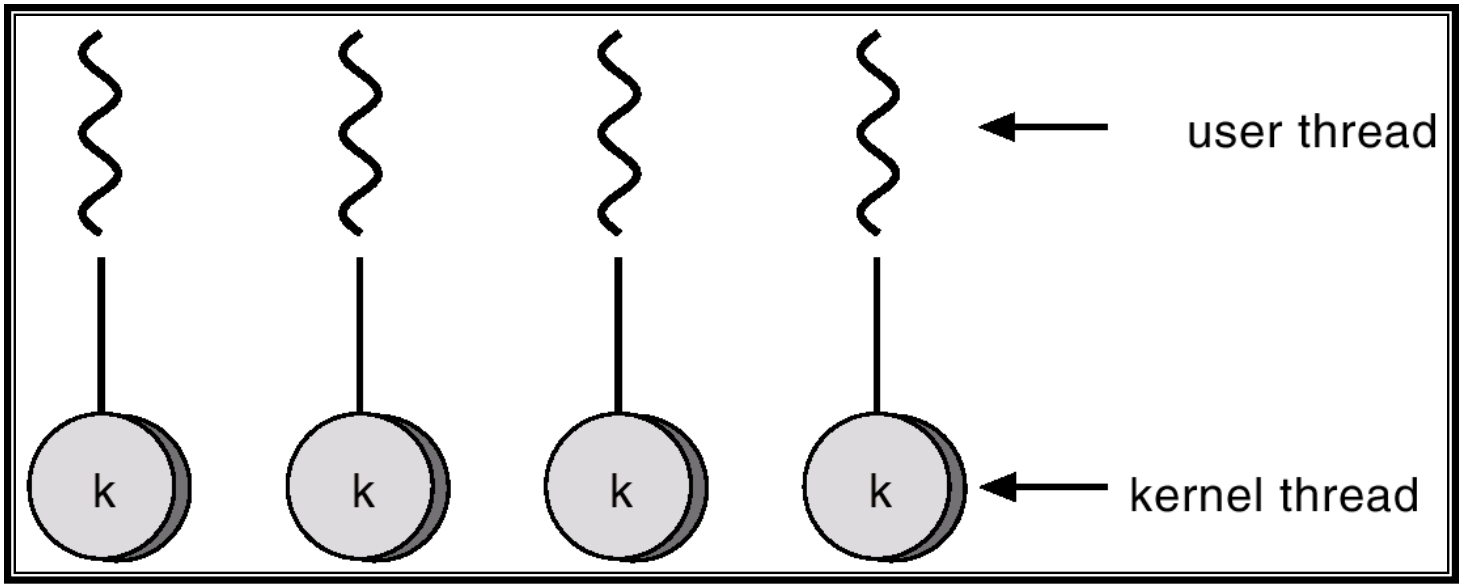
Kernel-Level Thread Implementation



- OS now manages threads and processes
  - All thread operations are implemented in the kernel
  - Kernel performs thread creation, scheduling, and management (each thread is a scheduling entity)
  - The OS schedules all of the threads in the system
- OS-managed threads are called kernel-level threads or lightweight processes
- Scheduler deals in threads
  - PCBs are no longer scheduled
  - If a thread blocks, another thread in the same process can run

Kernel Thread Implementation

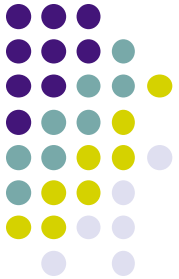
- Each user thread maps to a kernel thread
- Examples: Windows family, Linux



- Slow to create and manipulate

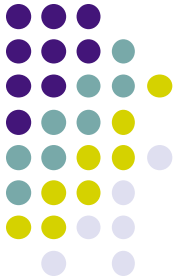
+ Integrated with OS well (e.g., a blocking syscall will not block the whole process)

Three multithreading models

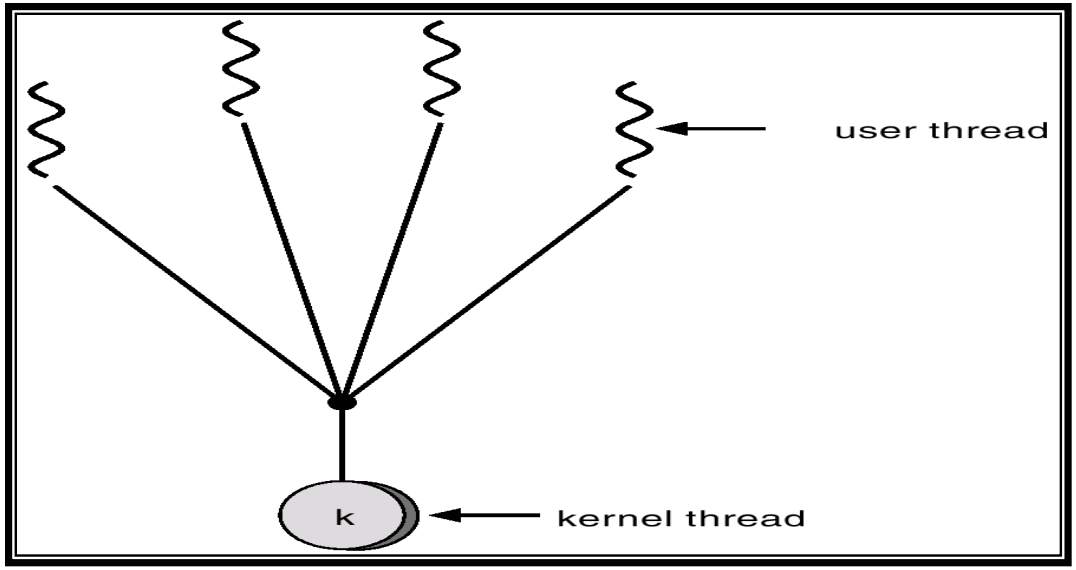


- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

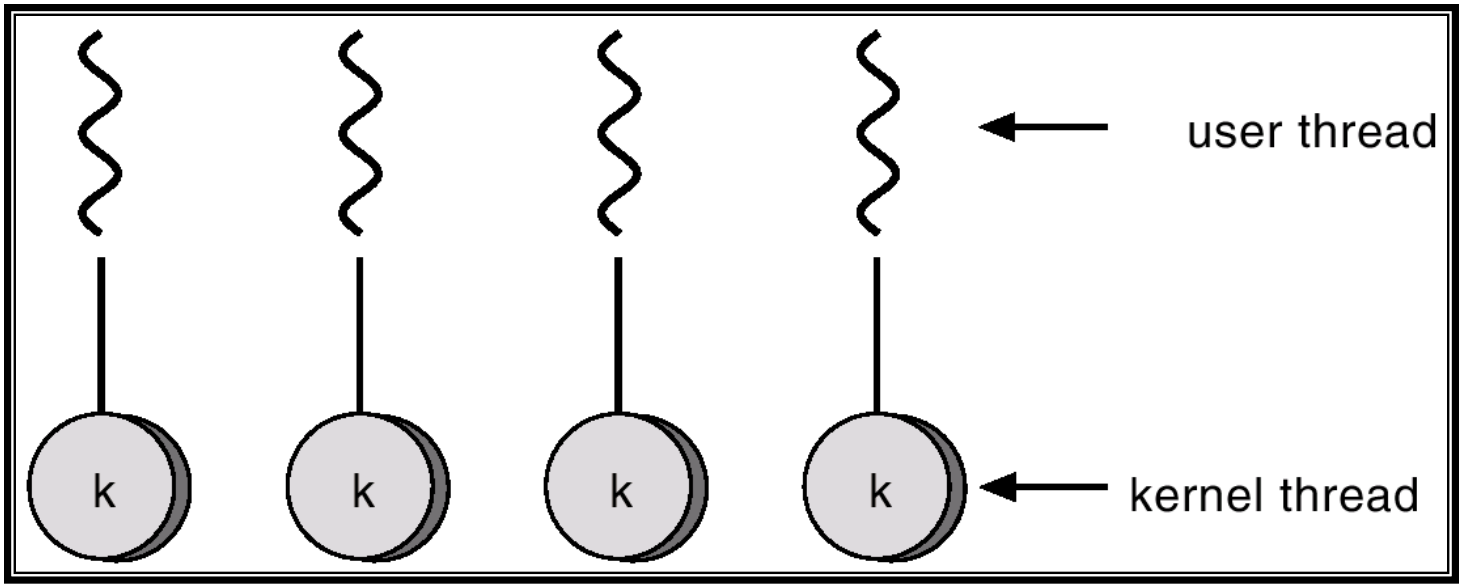


- Many user-level threads mapped to single kernel entity (kernel thread)
- Used in *user thread implementation*
- Drawback: blocking sys call blocks the whole process

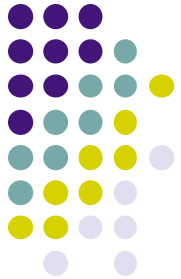


One-to-One

- Each user thread maps to kernel thread
- Used in *kernel thread implementation*
- May lead to too many kernel threads

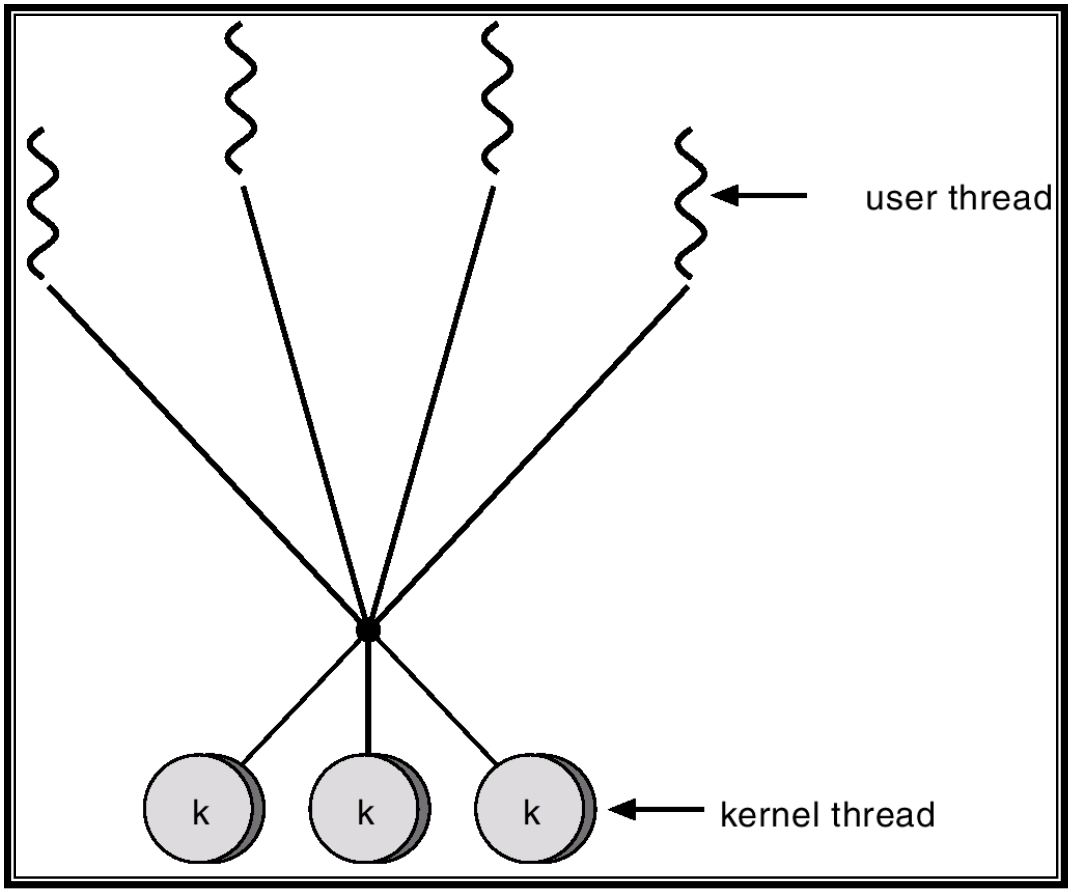


Many-to-Many model

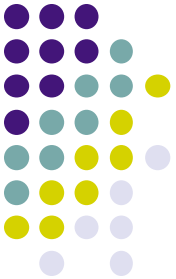


- Allows many user threads to be mapped to many kernel threads
- Allows OS to create a sufficient number of kernel threads running in parallel
  - When one blocks, schedule another user thread
- Examples:
  - Solaris 2
  - Windows NT/2000 with the *ThreadFiber* package

Many-to-Many Model

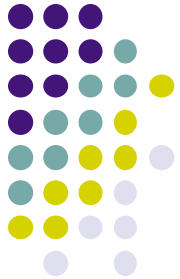


Context switching threads



- Context switching two user-level threads
  - If belonging to the same process
    - Handled by the dispatcher in the thread library
      - Only need to store/load the TCB information
    - OS does not do anything
  - If belonging to different processes
    - Like an ordinary context switch of two processes
      - Handled by OS (drop in/out of the kernel)
      - OS needs to load/store PCB information and TCB information

Web browser example



- Why do some modern web browsers want to use separate processes (e.g., for different tabs, extensions) instead of threads?

- Security isolation
- Reliability isolation