# Processes (and Threads)

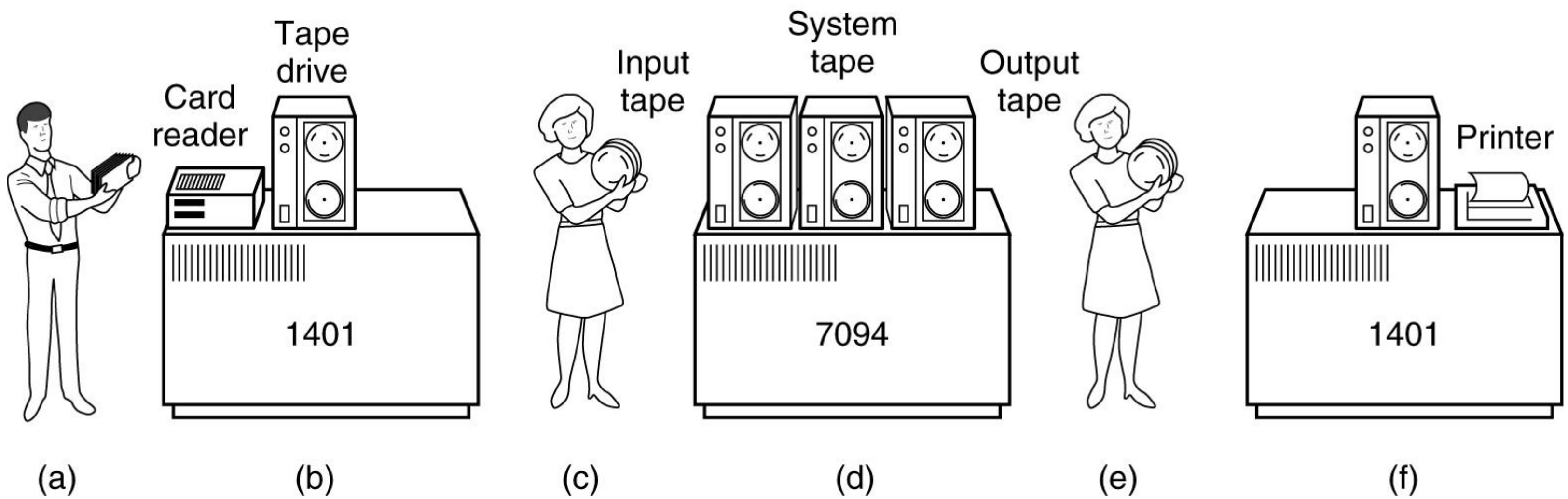
- Process Management
- Thread Models
- Inter-process Communication (IPC)
- Hierarchical Microkernel and IPC

# Processes: historical rationale

Manual start/stop of computational "tasks"

⇨ *Batch processing* (reduce task switching time)

⇨ *Multiprogramming* (reduce processor idle time)



# Process

- **Process:** A program in "execution"

  Program → *passive entity (specification)*
  Process → *active entity (execution of the specification)*

  *…with requisite data and resources*

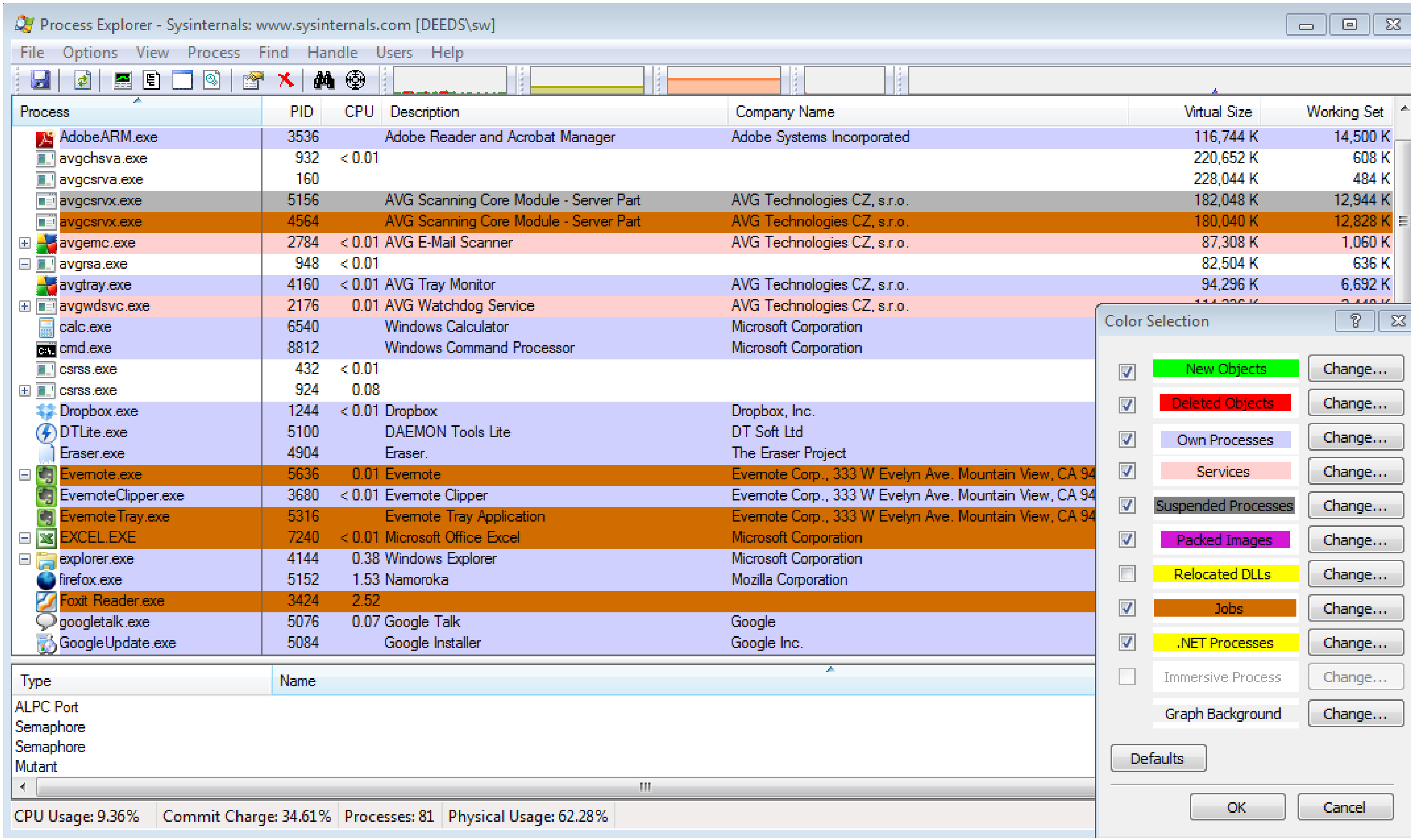
- Process creation & execution requires resources
  - processor, memory, files, …
  - Initialization data, I/O, …

- Processes need management

❖ Multiple processes (user, kernel) concurrently on one or more CPUs/cores ("multi-programming")

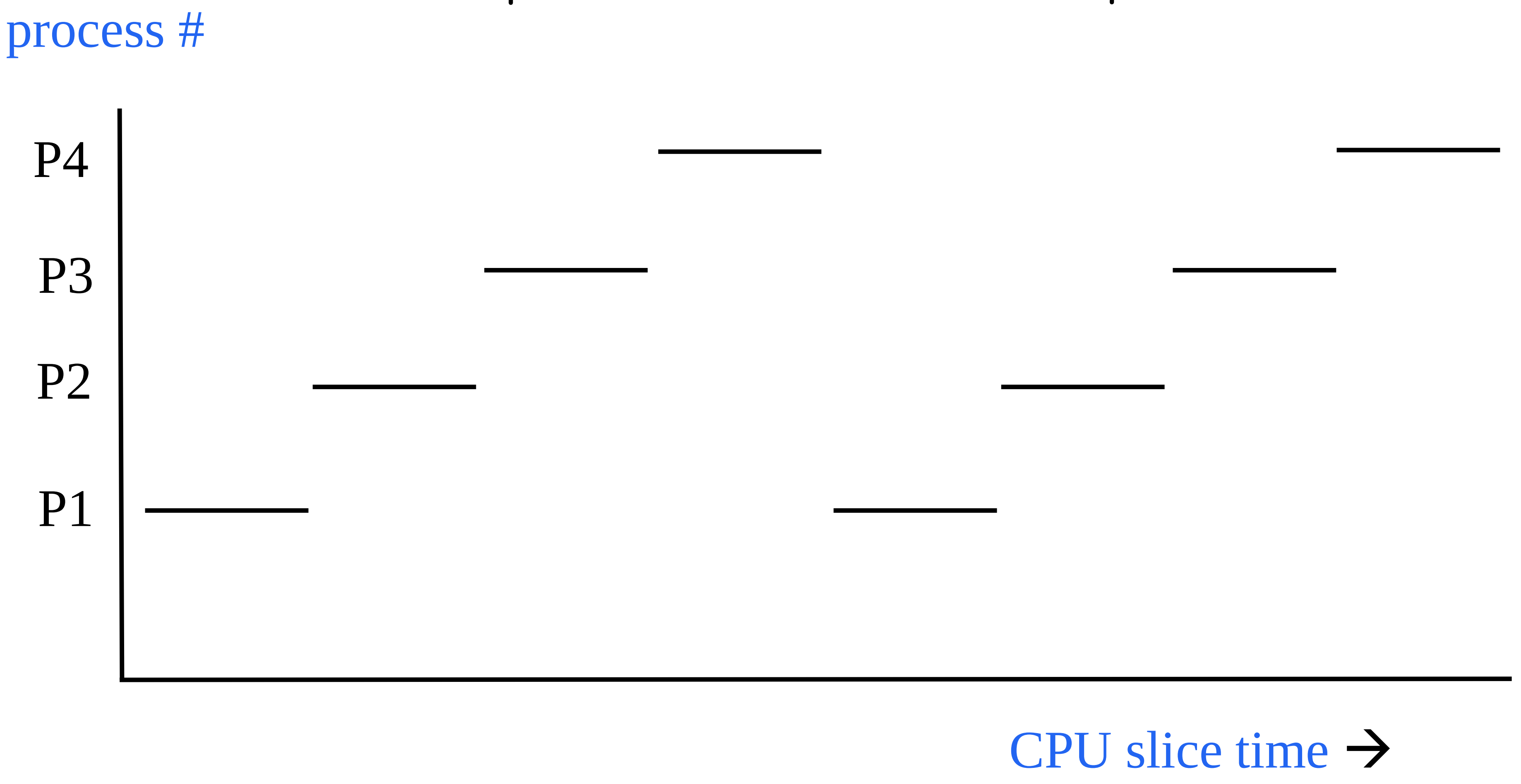
⇨ Limit degrees of interference, guarantee "fair" resource sharing

# Processes on a multiprogramming system

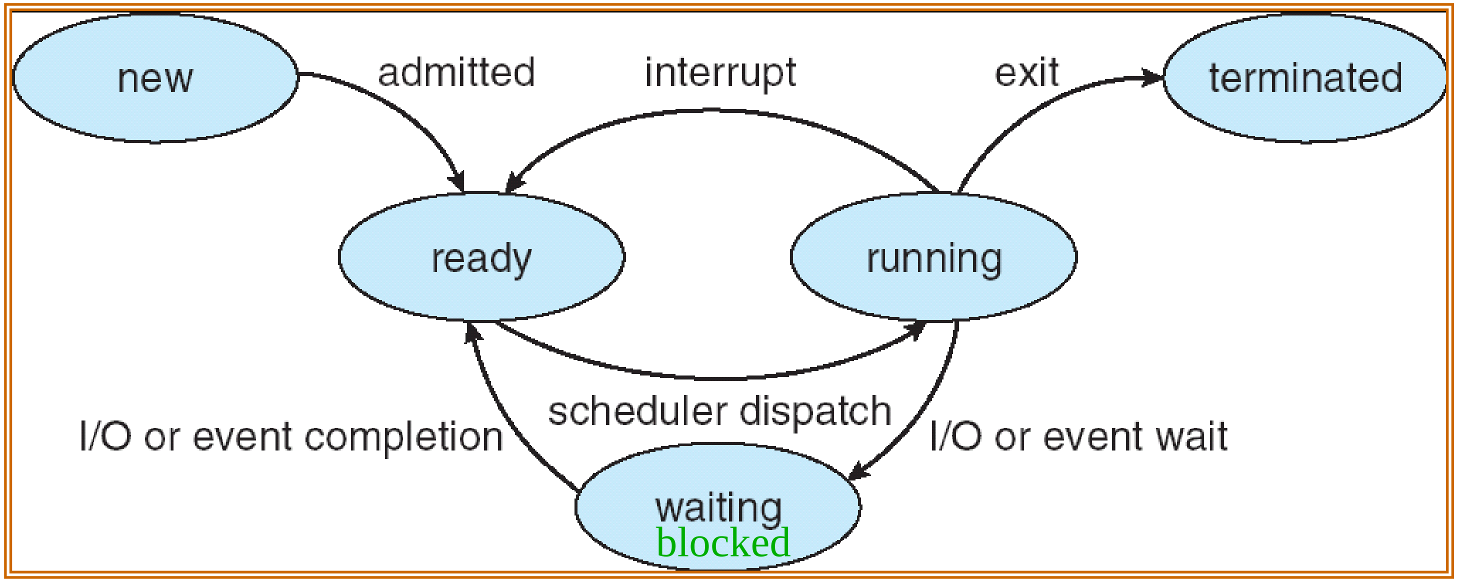


# Multiprogramming/Concurrent Processes

Each process' local execution is sequential!



# The Process States




© Silberschatz et al.: Operating System Concepts, 8th ed., Wiley

As a process executes, it changes ***state***:

- **new**: Process (parameters) initialized
- **ready**: Waiting to be assigned to CPU
- **running**: Instructions are being executed
- **waiting**: Waiting for data/I/O or events
- **terminated**: Finished execution

Processes Termination Conditions

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)

# Process states in Linux (from 2.6.26)

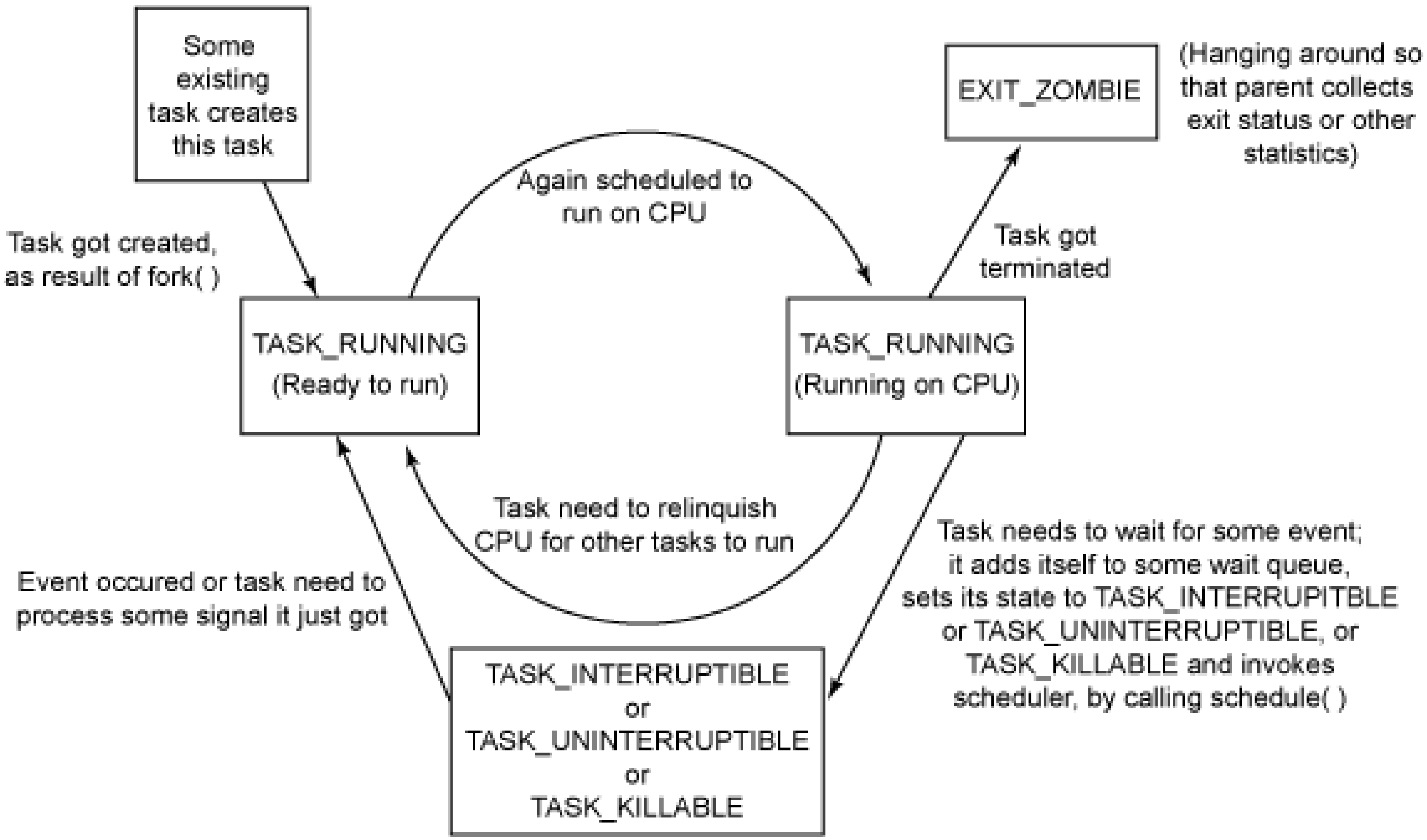



Figure (c) IBM: http://www.ibm.com/developerworks/linux/library/l-task-killable/

# OS →Process Management

1. Process resource allocation, handling, reclaiming
2. Running, suspending and resuming processes
3. Process creation
4. Process termination
5. Provide inter-process communication (IPC) for cooperating processes

- Process scheduling
- Process synchronization
- Process deadlock handling

# 1. Process resource allocation/handling/reclaiming

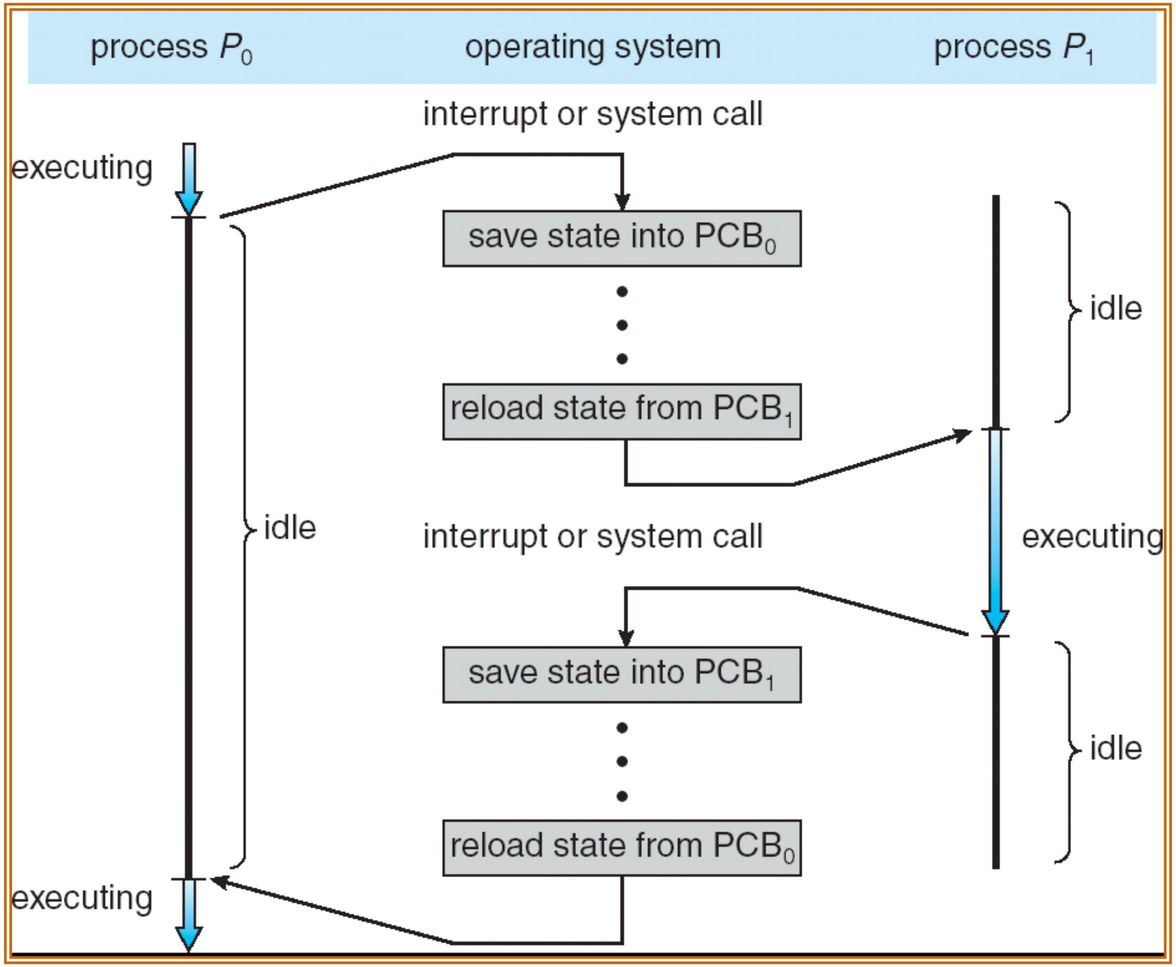
## Process Control Blocks (PCBs)

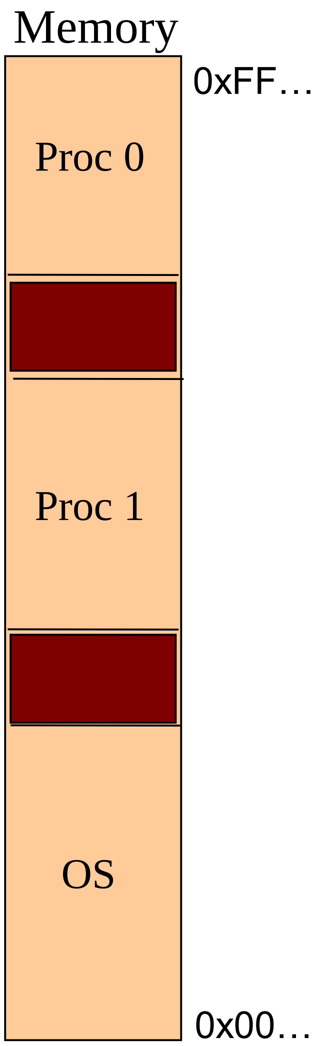
Process
==
virtual CPU?

| Process management | Memory management | File management |
|---|---|---|
| Registers | Pointer to text segment | Root directory |
| Program counter | Pointer to data segment | Working directory |
| Program status word | Pointer to stack segment | File descriptors |
| Stack pointer | | User ID |
| Process state | | Group ID |
| Priority | | |
| Scheduling parameters | | |
| Process ID | | |
| Parent process | | |
| Process group | | |
| Signals | | |
| Time when process started | | |
| CPU time used | | |
| Children's CPU time | | |
| Time of next alarm | | |

# 2. Running, suspending and resuming processes

Each process' local execution is sequential!




Context-switch time is "overhead" (on OS + HW)
the system does no useful work while switching

© Silberschatz et al.: Operating System Concepts, 8th ed., Wiley

# 3. Process creation

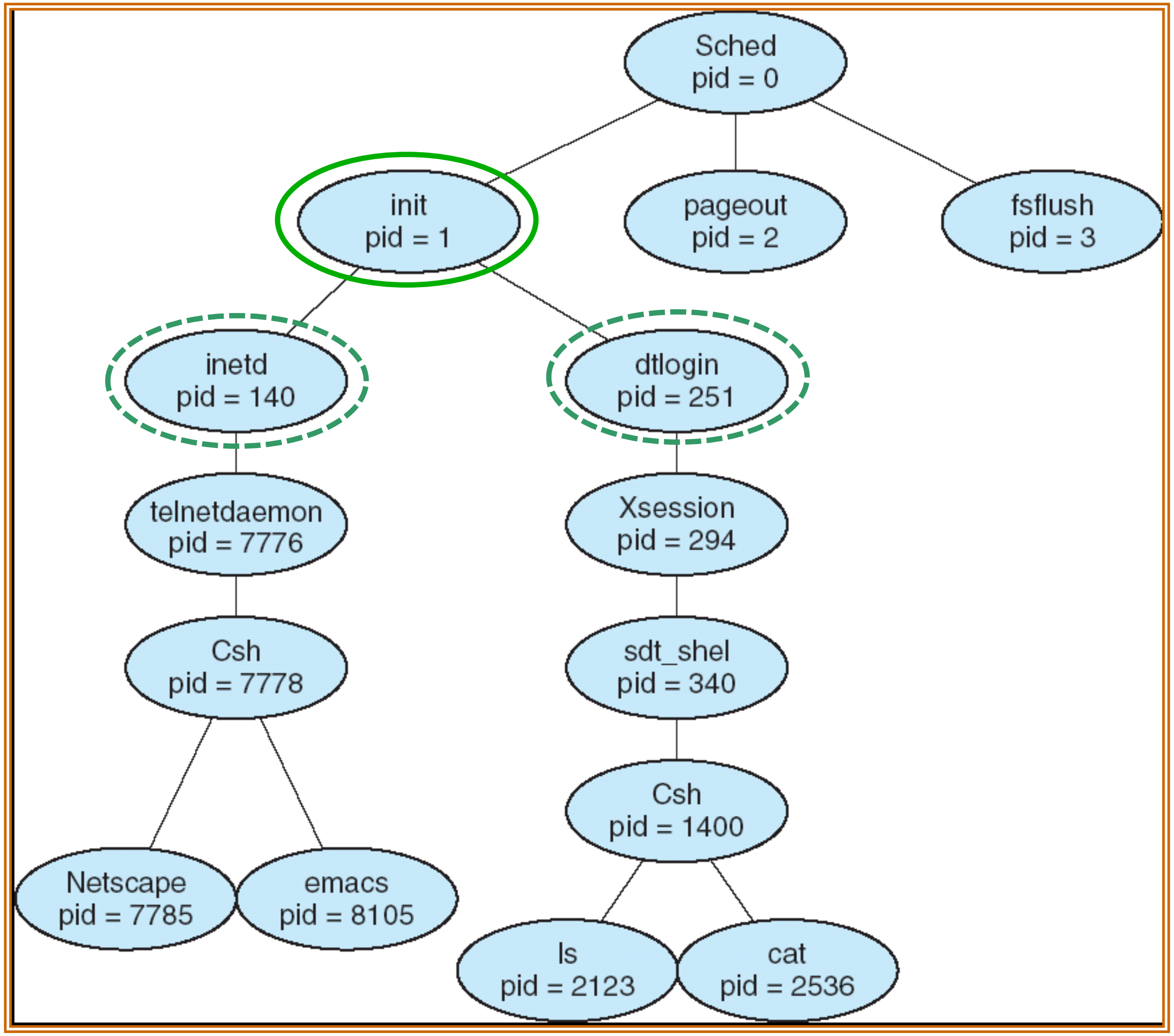
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
(inter-process naming & ordering considered later)

- Issue/Options: Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Issue/Options: Concurrent/sequential execution
  - Parent and children execute concurrently
  - Parent waits until some/all children terminate
- Issue/Option: Program to execute
  - Child is a duplicate of parent
  - Child has another program loaded into it

- UNIX
  - **fork** system call creates new (clone) process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

# Tree of processes: Solaris

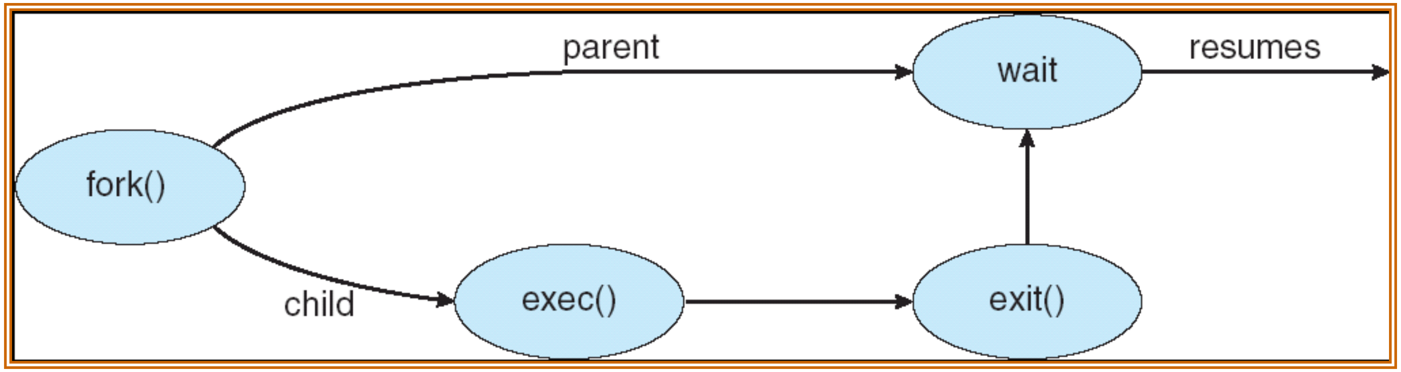
Each process has a unique ID called PID



# Process hierarchies

- Parent creates a child process; child processes *can* become a standalone process (different program, state, possibly sharing memory/files)
- Parent/child relation results in hierarchy
  - **UNIX** calls this a "process group"
    - Parent – child relation cannot be dropped
    - Parent – child maintain distinct address spaces; initially child inherits/shares parent's address space contents
  - **Windows** has a different concept of process hierarchy
    - processes can be created sans implicit heritage relations (though a parent can control a child using a "handle")
    - clean address space from start

# Process creation (POSIX)




© Silberschatz et al.: Operating System Concepts, 8th ed., Wiley

- **fork** system call creates new (clone) process
- **exec** system call used after a **fork** to replace the process' memory space with a new program
- parent "waits" till child finishes execution

# C Program: Forking separate process (POSIX)

```c
int main()
{
    pid_t  pid;

    pid = fork();          /* fork a child process */
    if (pid < 0) {         /* error occurred */
       fprintf(stderr, "Fork Failed");
       exit(-1);
    }
    if (pid == 0) {        /* child process */
       execlp("/bin/ls", "ls", NULL);
    }
    else {                 /* parent process */
       wait (0);           /* parent will wait for the child to complete */
       printf ("Child Complete");
       exit(0);
    }
}
```

# 4. Process termination

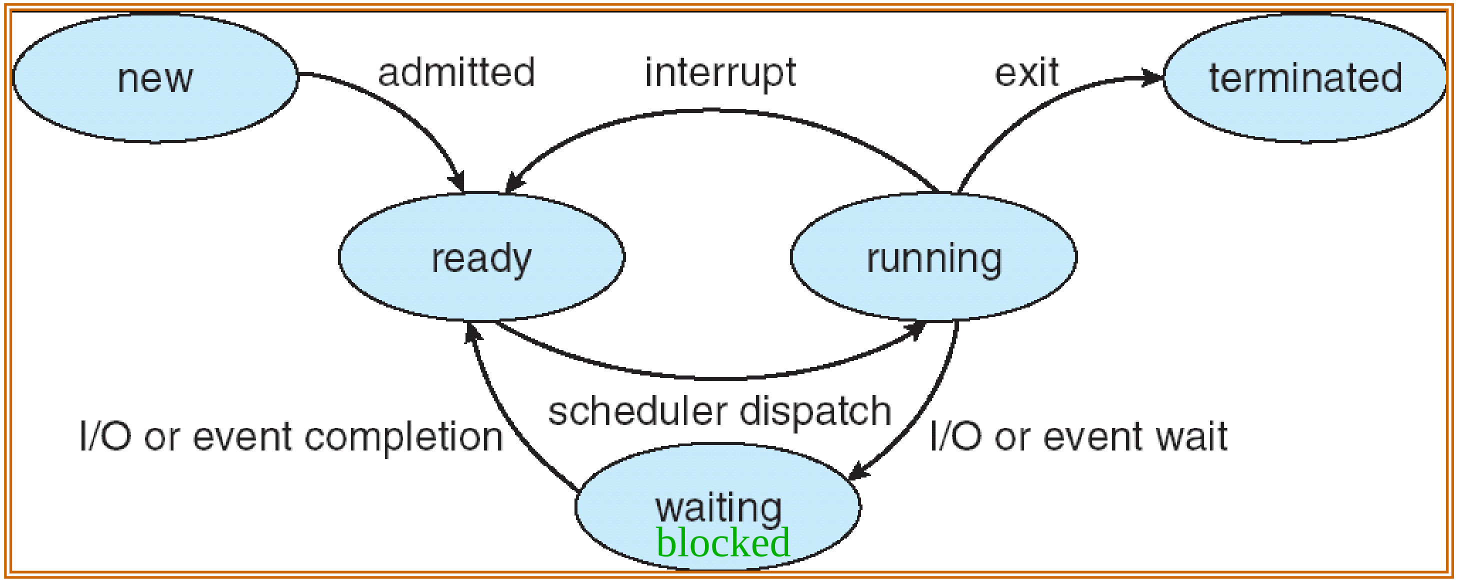
- Process executes last statement and asks OS to delete it → **exit()**
  - Return status value to parent (via **wait**)
  - Process' resources are de-allocated by OS

- Parent may terminate execution of children processes (**kill()**, **TerminateProcess()**) if
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates (zombie control)
      - *All children terminated - *cascading termination*

# Processes (and Threads)

- Process Management
- <u>Thread Models</u>
- Inter-process Communication (IPC)
- Hierarchical Microkernel and IPC

# Recap: Process States




© Silberschatz et al.: Operating System Concepts, 8th ed., Wiley

As a process executes, it changes ***state***:

- **new**: Process (parameters) initialized
- **ready**: Waiting to be assigned to CPU
- **running**: Instructions are being executed
- **waiting**: Waiting for data/I/O or events
- **terminated**: Finished execution

Processes Termination Conditions

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)

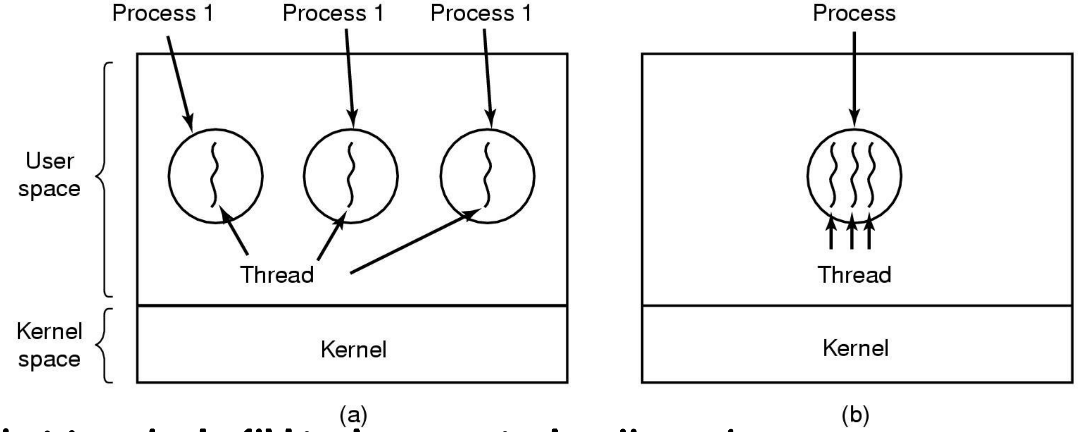
# OS: Concurrency?

- OS: Multiple activities & multiple resources
  … typical 80-20% I/O-CPU usage basis
  - Can we maximize execution & resource utilization via concurrency?
  - Can we decouple dispatching (process resource set up etc) and execution activities?

- Can we go from monolithic "process/children" style sequential abstractions to…
- simpler & faster "sub-process" activities (for execution and programming) via a "Divide and Conquer" parallelization approach?
  → **Threads**

# Threads: Flow Control within a Process

- Process Model (heavyweight single thread)
  - Each process has discrete + distinct (sequential) control flow
  - Each process/child has its unique PC, SP, registers + address space
  - Processes interact via IPC




© A.S.Tanenbaum: Modern Operating Systems, 3rd ed., Pearson/Prentice Hall

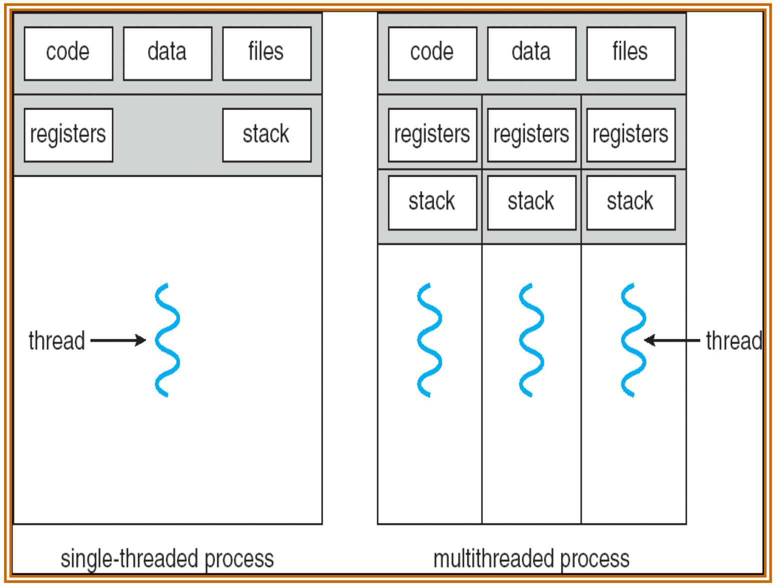
- Thread Model ("lightweight" sub-processes)
  - Each thread runs independently though sequentially (like a process)
  - Each thread has its own PC, SP (like a process)
  - A thread can spawn sub-threads (like a process)
  - A thread can request services (like a process)
  - A thread has "state" ready:running:blocked (like a process)

# **But**, all threads share exact same address space

- access to all global variables within the process
- access to all files within the shared address space
- can read, write, delete variables, files and stacks

+ simpler/faster
+ concurrency!!!
- no isolation
  security?

shared address space




© Silberschatz et al.: Operating System Concepts, 8th ed., Wiley

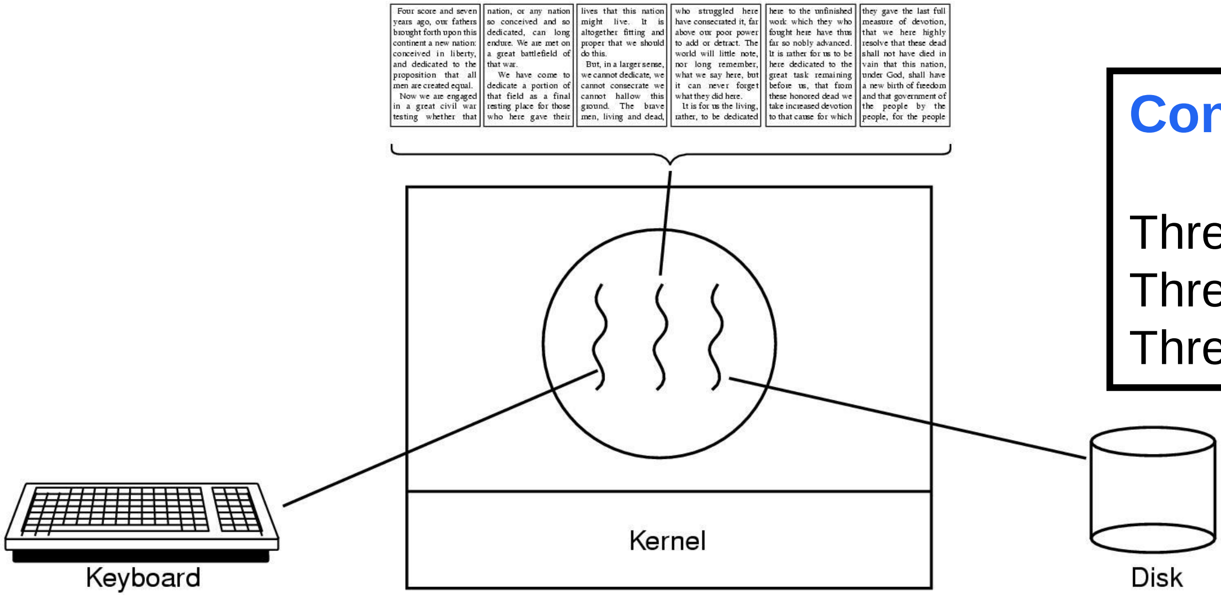
Shared "Owner" Process Info

| **Per process items** | **Per thread items** |
|---|---|
| Address space | Program counter |
| Global variables | Registers |
| Open files | Stack |
| Child processes | State |
| Pending alarms | |
| Signals and signal handlers | |
| Accounting information | |

Private Thread Execution Info

"simple & local" **Thread Table**

# Concurrency with shared address space




© A.S.Tanenbaum: Modern Operating Systems, 3rd ed., Pearson/Prentice Hall

**Concurrent Activities**

Thread 1: CPU: Formatting
Thread 2: I/O (keyboard, display)
Thread 3: Storage

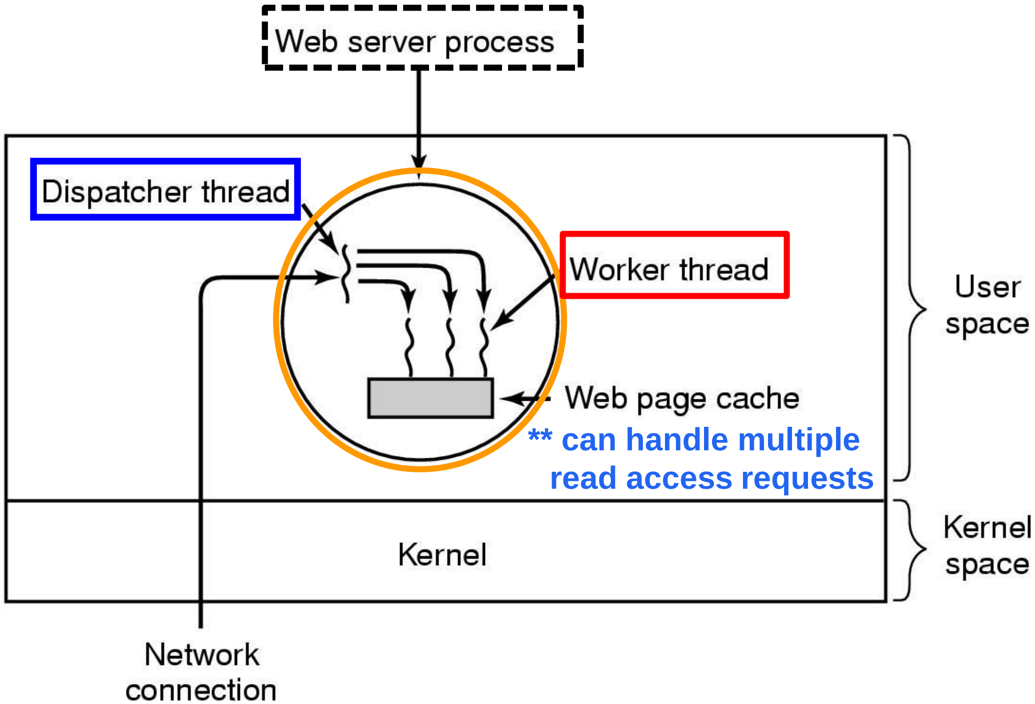
**Multiple concurrent tasks with different resource needs**

Non-blocking decoupled executions possible via shared file access, shared address space…

Viable with processes/children that have distinct address spaces?

# Thread Usage (Multi-threaded Web Server)

**Request Handling decoupled from Request Execution: concurrency, performance..**




© A.S.Tanenbaum: Modern Operating Systems, 3rd ed., Pearson/Prentice Hall

**Dispatcher** (dispatch & forget)

```
while(TRUE){
        get_next_request(&buf);
        hand_off_work(&buf);}
```

**Worker** (loop till done)

```
while(TRUE){
 wait_for_work(&buf);
 look_for_page_in_cache(&buf,&page);
 if(page_not_in_cache(&page)){
  read_page_from_disk(&buf,&page);
  }
return_page(&page);}
```

* multi-process/child model would also work but entails much higher overhead: process creation, context switching, scheduling, discrete address spaces, discrete resource allocation etc

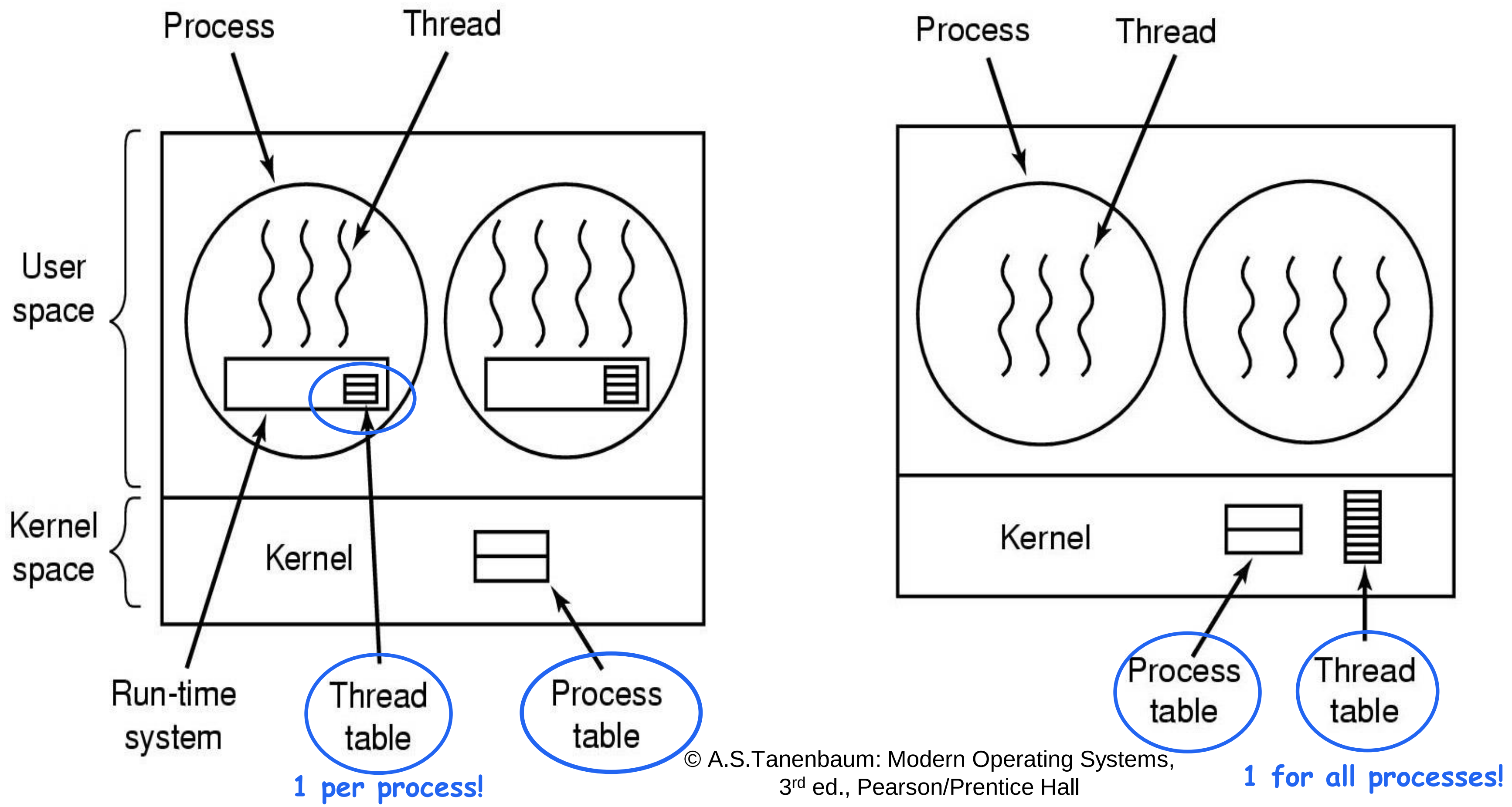
# Threading Comments

1. Responsiveness
   - Allows a program to continue even if parts of it are blocked
   - Ex: Tabs in Firefox, Opera, Text/Image Web server streams etc.
2. Resource Sharing (but also less protection!)
   - Threads share memory and process resources
   - Allows app. to perform several different activities on the same data
3. Efficiency/Performance
   - More economical to context-switch threads than processes
   - Solaris: 30-100 times faster thread creation vs. process creation; context switch 5 times faster for threads vs. processes
4. Utilization of multiprocessor architectures
   - Worker threads dispatching to different processors

- Joint process/thread schedulers – complex
- Complex resource sharing/ordering, termination issues etc

Implementations:

## User-level | Kernel-level




© A.S.Tanenbaum: Modern Operating Systems, 3rd ed., Pearson/Prentice Hall

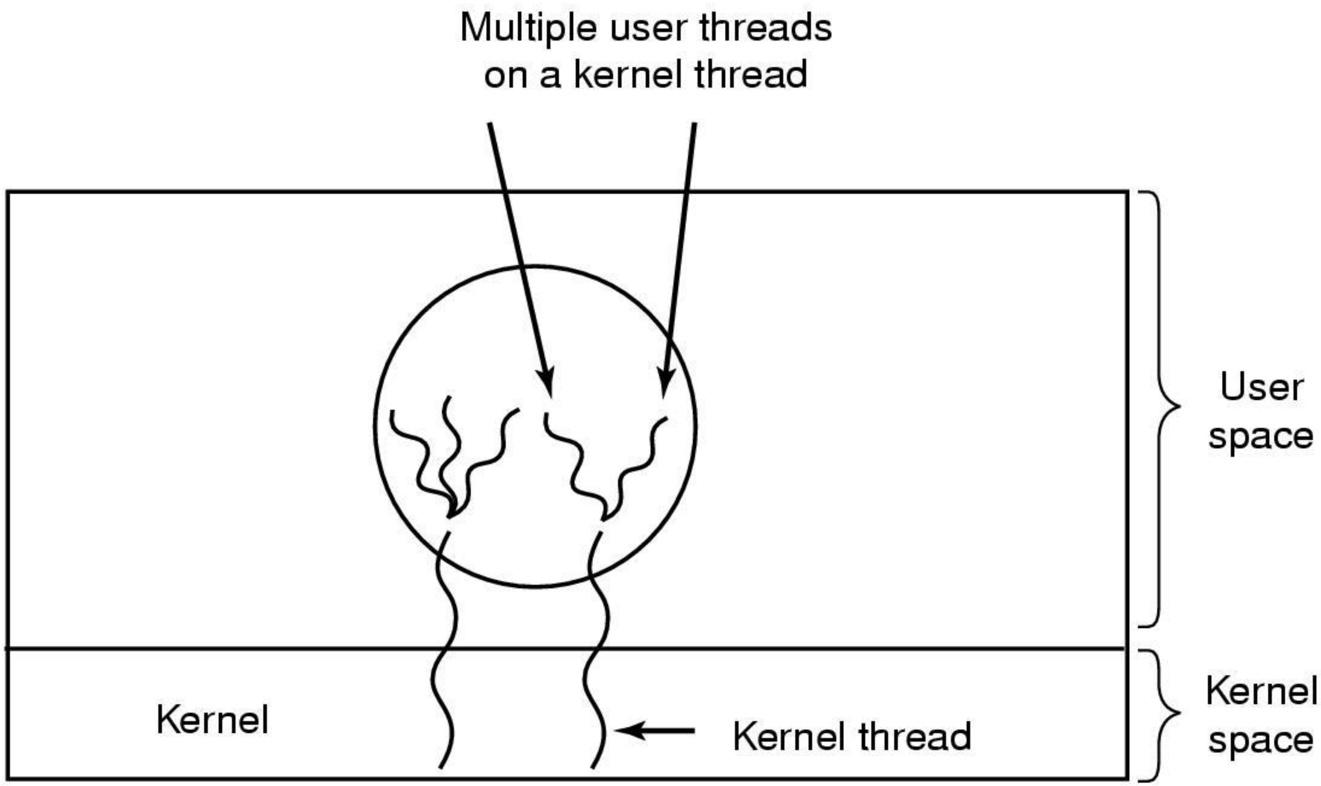
## User-level threads package

+ each user process can define its thread policies!
+ flexible localized scheduling
- NO kernel knowledge ➔ no kernel support for thread management

## Kernel managed threads package

+ single thread table under kernel control
+ full kernel overview and thread management

# Hybrid Implementations

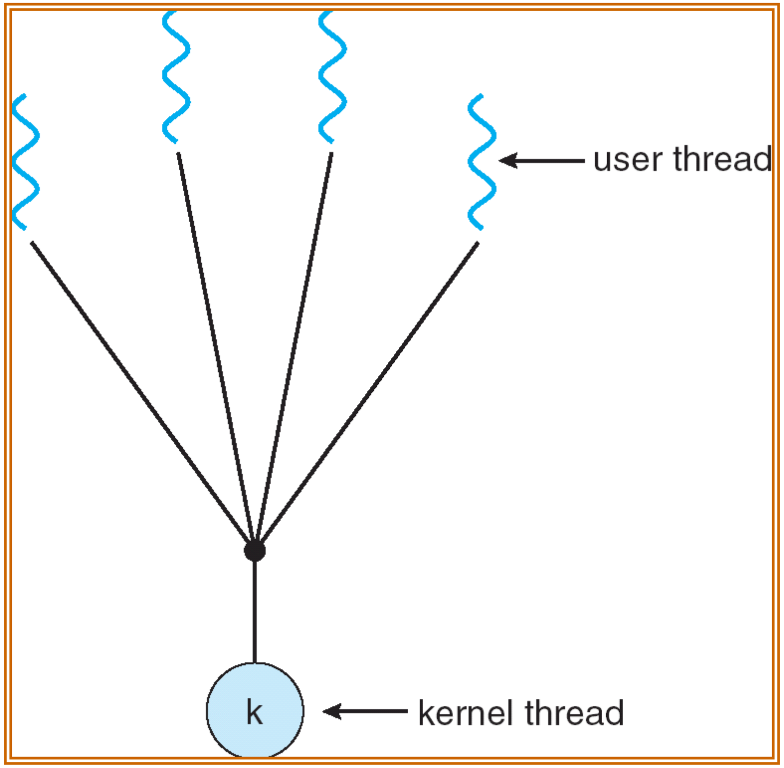



© A.S.Tanenbaum: Modern Operating Systems, 3rd ed., Pearson/Prentice Hall

Multiplexing user-level threads onto kernel-level threads
(each kernel thread possesses limited sphere of user-thread control)

# 1. Many-to-One Model

- Many user-level threads mapped to single kernel thread
  - Solaris Green Threads
  - GNU Portable Threads

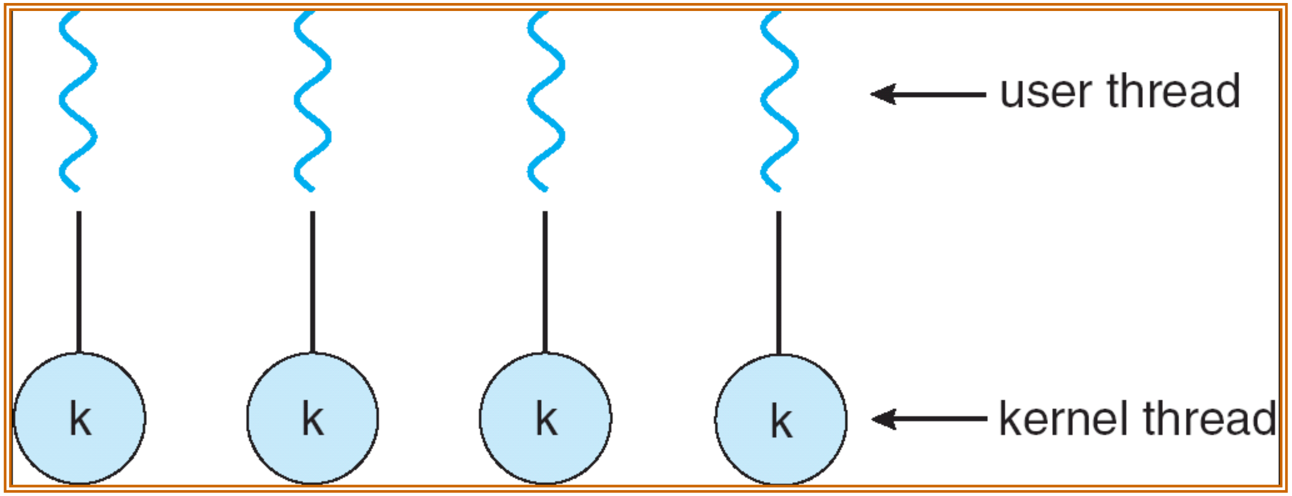



+ Flexible: thread mgmt. in user space

- Process blocks if thread blocks
- Only 1 thread can access kernel
  at a time: no multi-proc support

# 2. One-to-One Model

- Each user-level thread maps (bound) to a kernel thread
  - Windows
  - Linux
  - Solaris 9 and newer




+ Max. concurrency

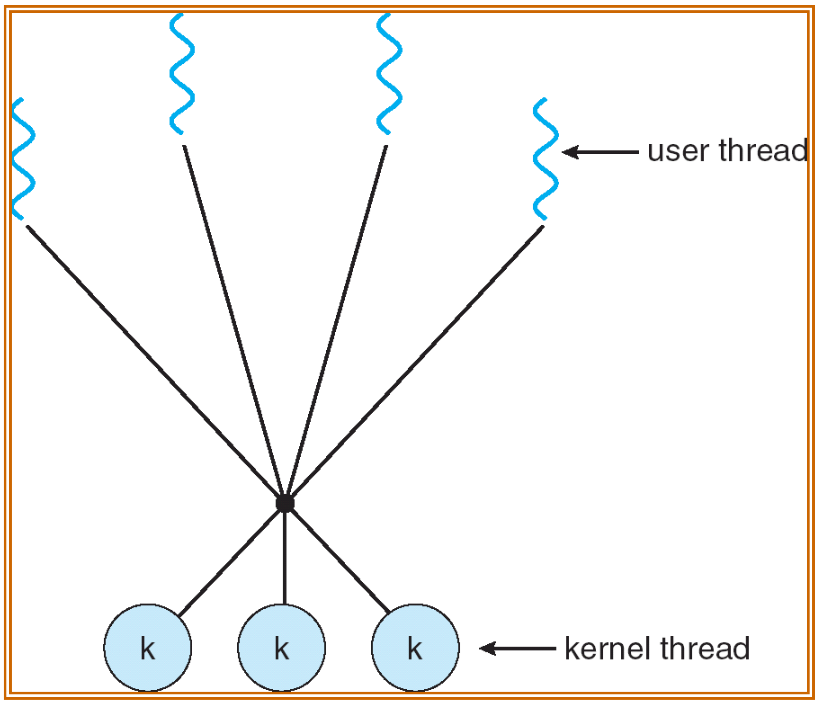
- Each user-thread needs kernel-thread (overhead!)

# Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Linux does not distinguish between process/threads
- Thread creation is done through **clone()** system call with flags indicating sharing across parent/children
- **clone()** allows a child task to share the address/file/signal space of the parent task (process)

# 3. Multiplexed: Many-to-Many Model*

- Many user level threads to be mapped to many kernel threads
- Allows OS to create/manage limited kernel threads
- Solaris (prior to v9); v9+ → one-to-one
- Windows NT family: with the *ThreadFiber* package

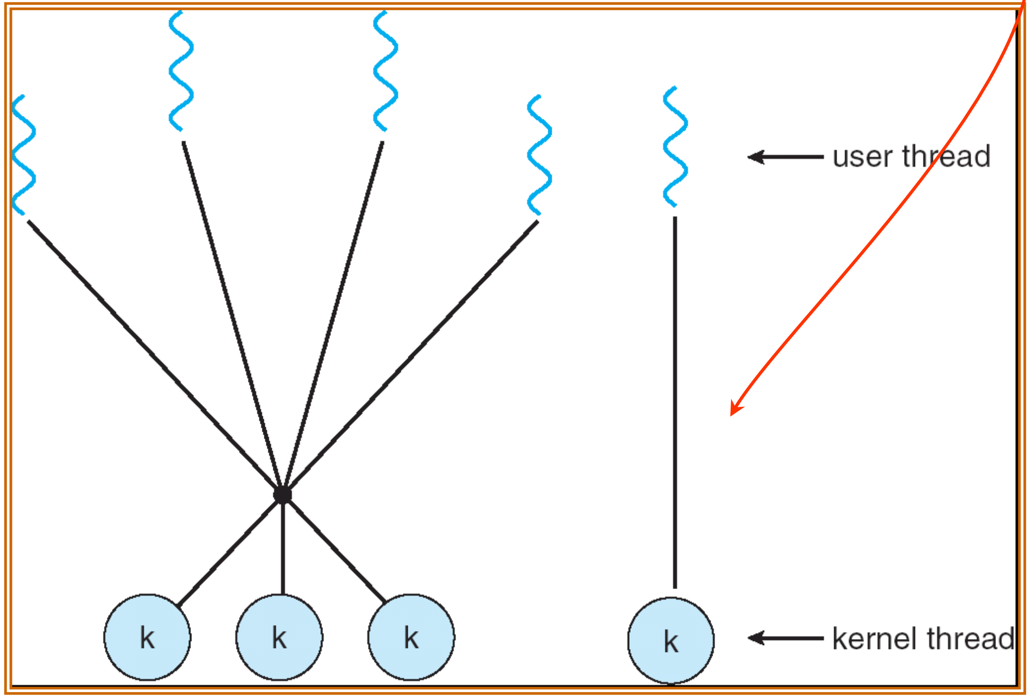



+ large # of threads
+ user-thread blocks, kernel schedules another for execution

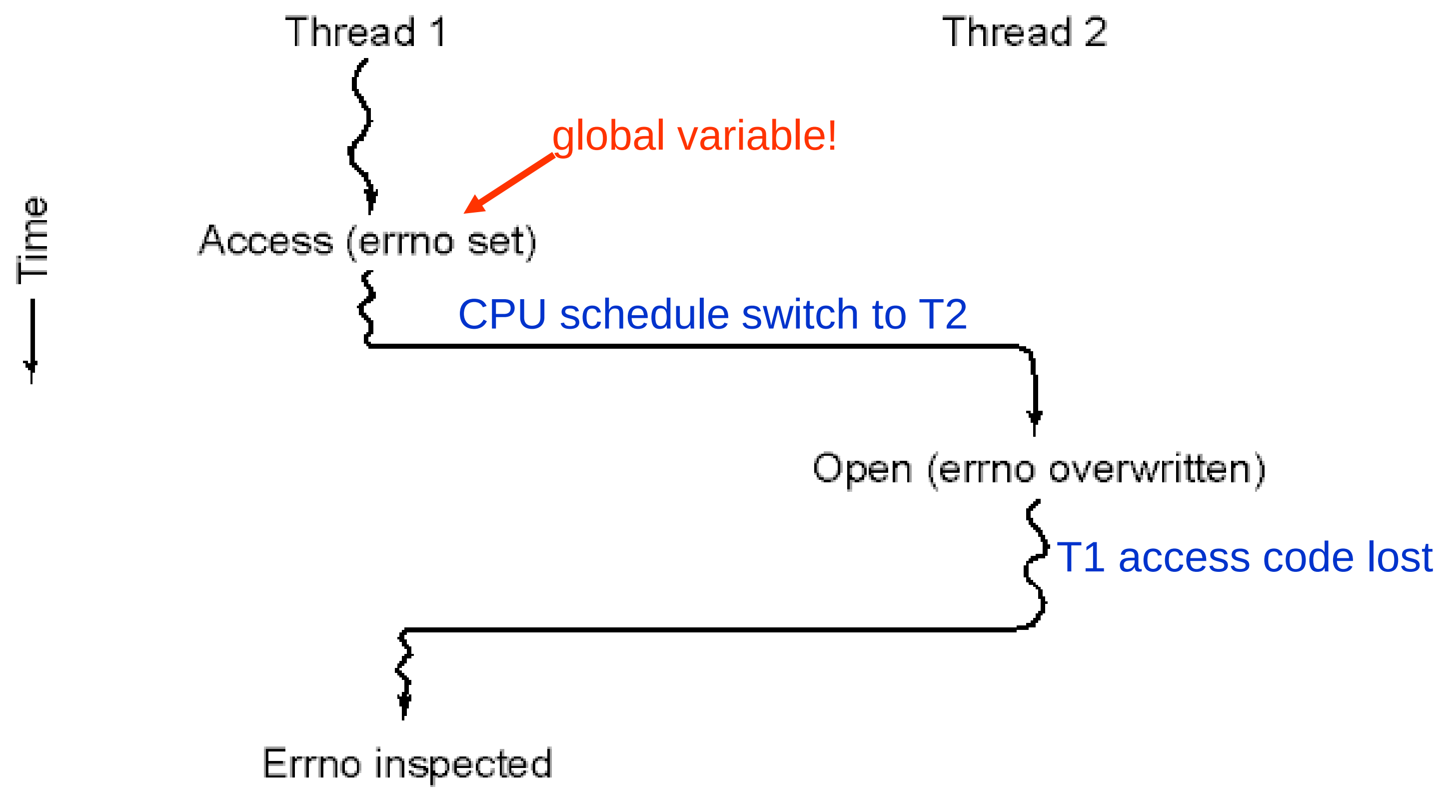
- lesser concurrency than 1-1 but easier scheduler and diff. k.threads for different server types

# 4. Two-level Model*

- Similar to M:N, except that it also allows a user thread to be **bound** to a kernel thread
  - HP-UX
  - 64bit UNIX
  - Solaris 8 and earlier



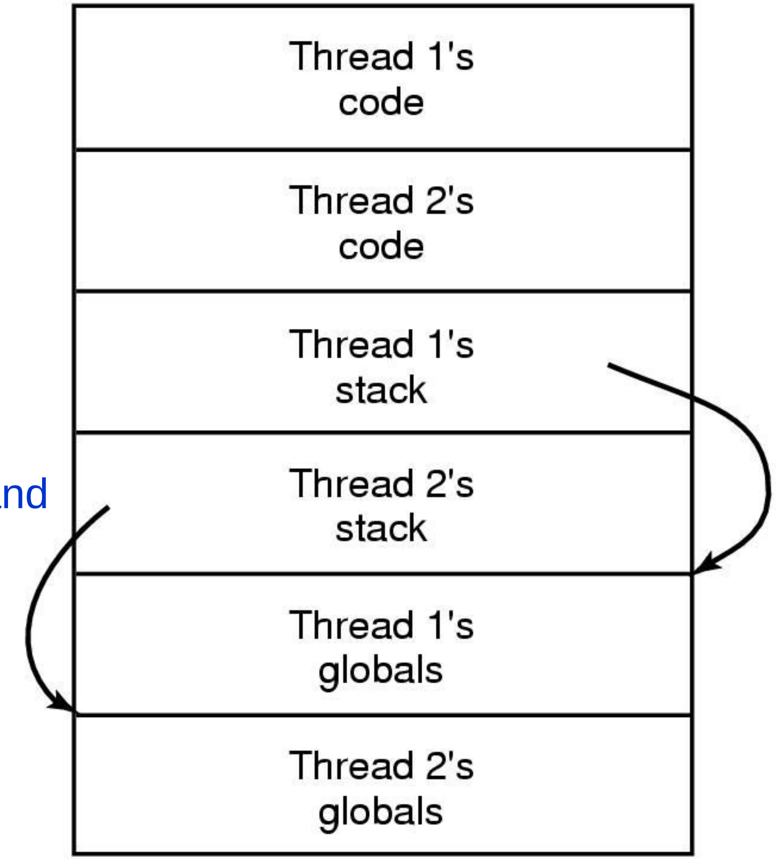
# Making Single-Threaded Code Multithreaded




Conflicts between threads over the use of a global variable

# Making Single-Threaded Code Multithreaded

* Avoid global variables completely?
* Allow threads to have private global variables




Each thread keeps private copy of errno and global variables

# Thread Mgmt: Processes → Threads → ?

- # of resources in a system: finite
- Traditional Unix (life was easy!)
  - single thread of control
  - multiprogramming: 1 process actually executing
  - non-preemptive processes

- Distributed systems, multi-threading etc.
- How do we handle issues of:
  - resource constraints
  - ordering of processes/threads
  - precedence relations
  - access control for global parameters
  - shared memory
  - IPC
  - Scheduling etc.

such that the solutions are: fair, efficient, deadlock and race-free?