Operating Systems – Threads

ECE 344 – Week 3

Does OS/161 support threads?

Major OS Requirements

- **Interleave** the execution of several processes
  - **Maximize** CPU **utilization** while maintaining reasonable response time
  - Support multiple users working **interactively**
  - **Convenience** (edit one program, while compiling another one)
- **Allocate** available **resources** for the execution of programs
- Provide for **communication facilities** between executing programs

Processes Revisited

Several definitions of process:

- A program in execution; a ***thread of execution***
- A unit of ***resource ownership***
- …

Many applications consist of **more than one thread of execution** which share resources

- Not supported in our current view of processes

Processes vs. Threads

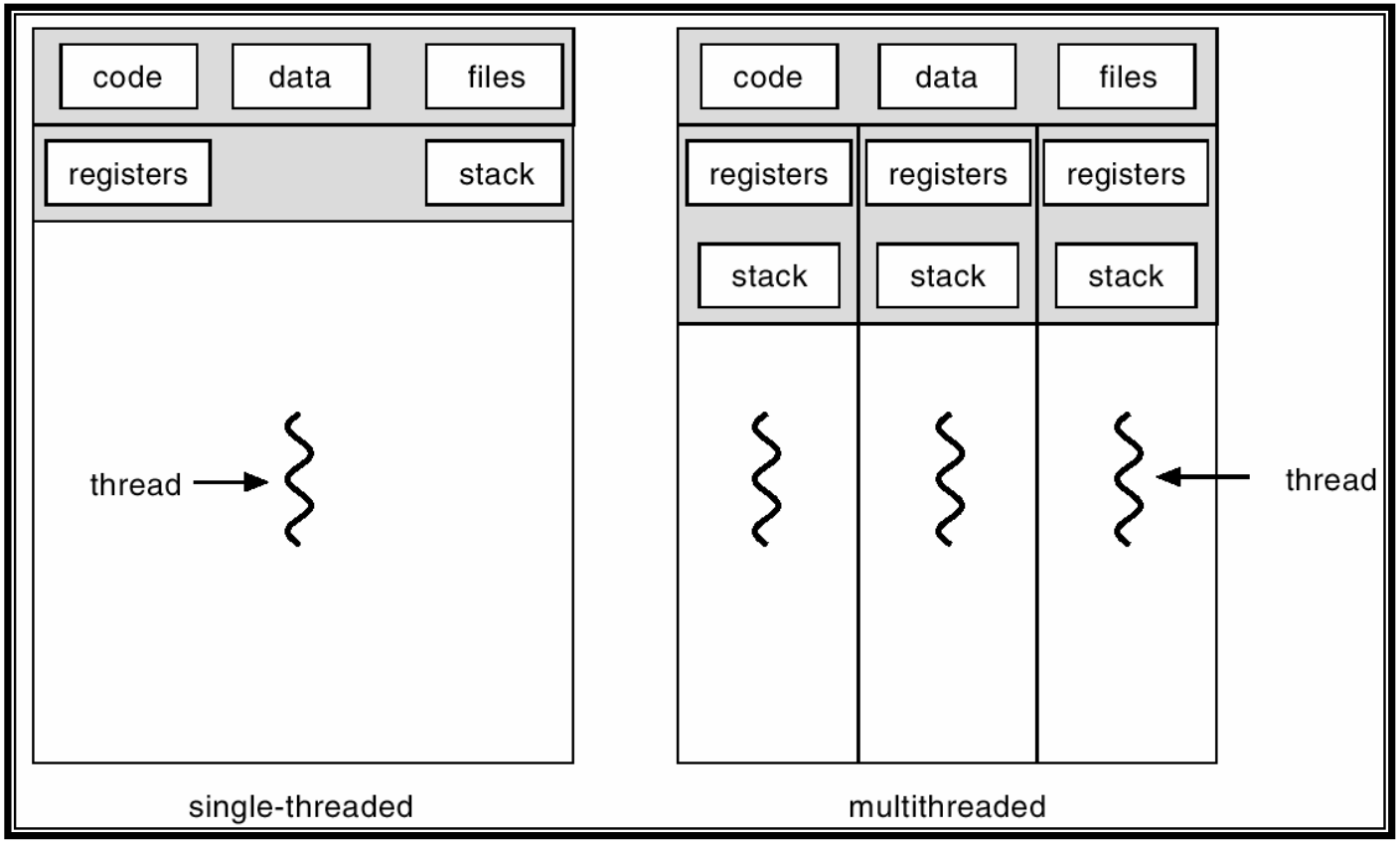
Process

- **Unit of resource ownership** with respect to the execution of a single program
- Can encompass **more than one thread of execution**
  - E.G., Web browser/server evolution: more than one thread for calendar, GUI, mail (browser) and handling requests (server)

Thread

- **Unit of execution**
- Belongs to a process
- Can be traced (i.e.., list the sequence of instructions)

Single and Multithreaded Processes



Terminology

- Lightweight process (LWP), threads, multithreaded processes

Shared and Private Items of Threads

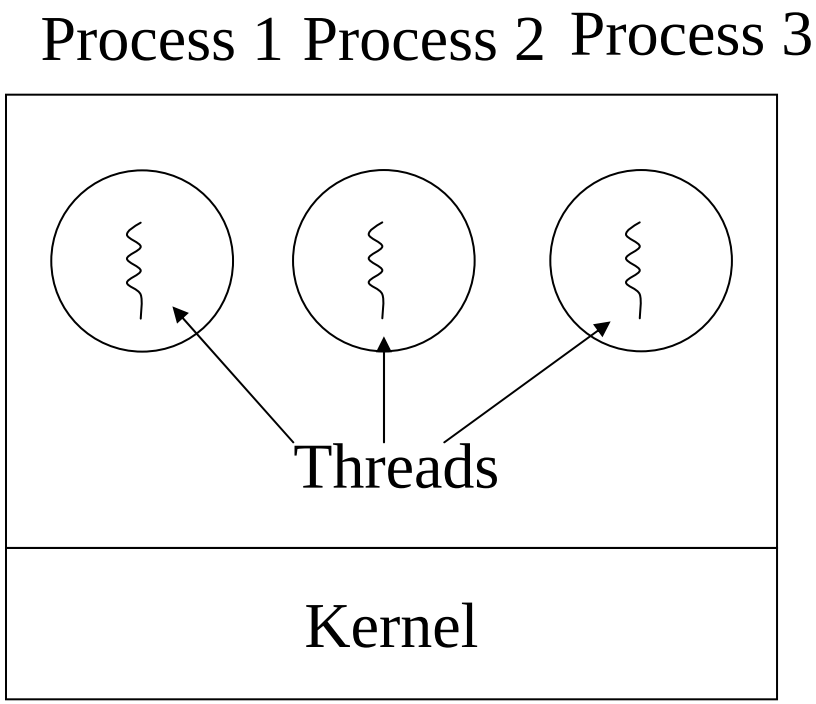
Per Process items

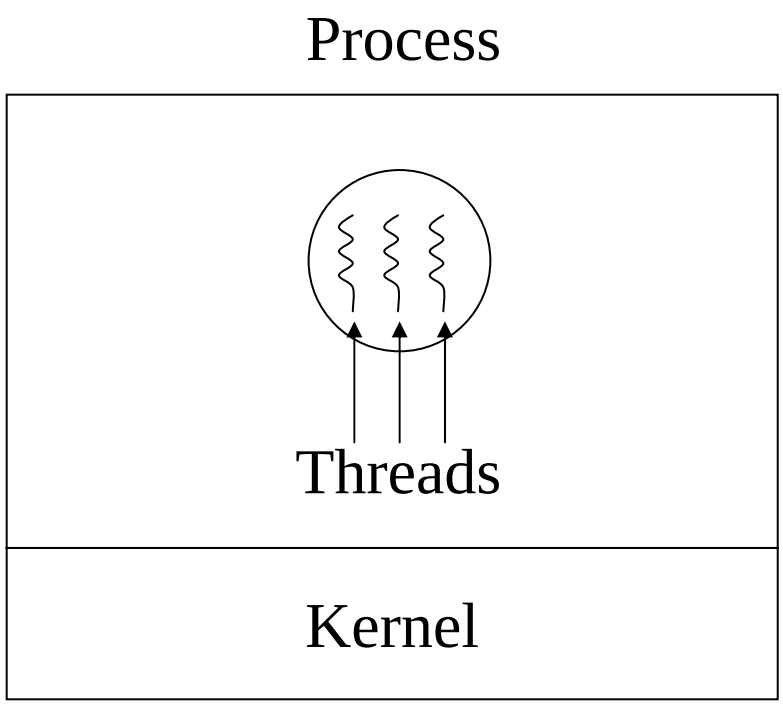
- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signal and signal handlers
- Accounting information

Per thread items

- Program counter
- Registers
- Stack
- Thread state

Processes vs. Threads





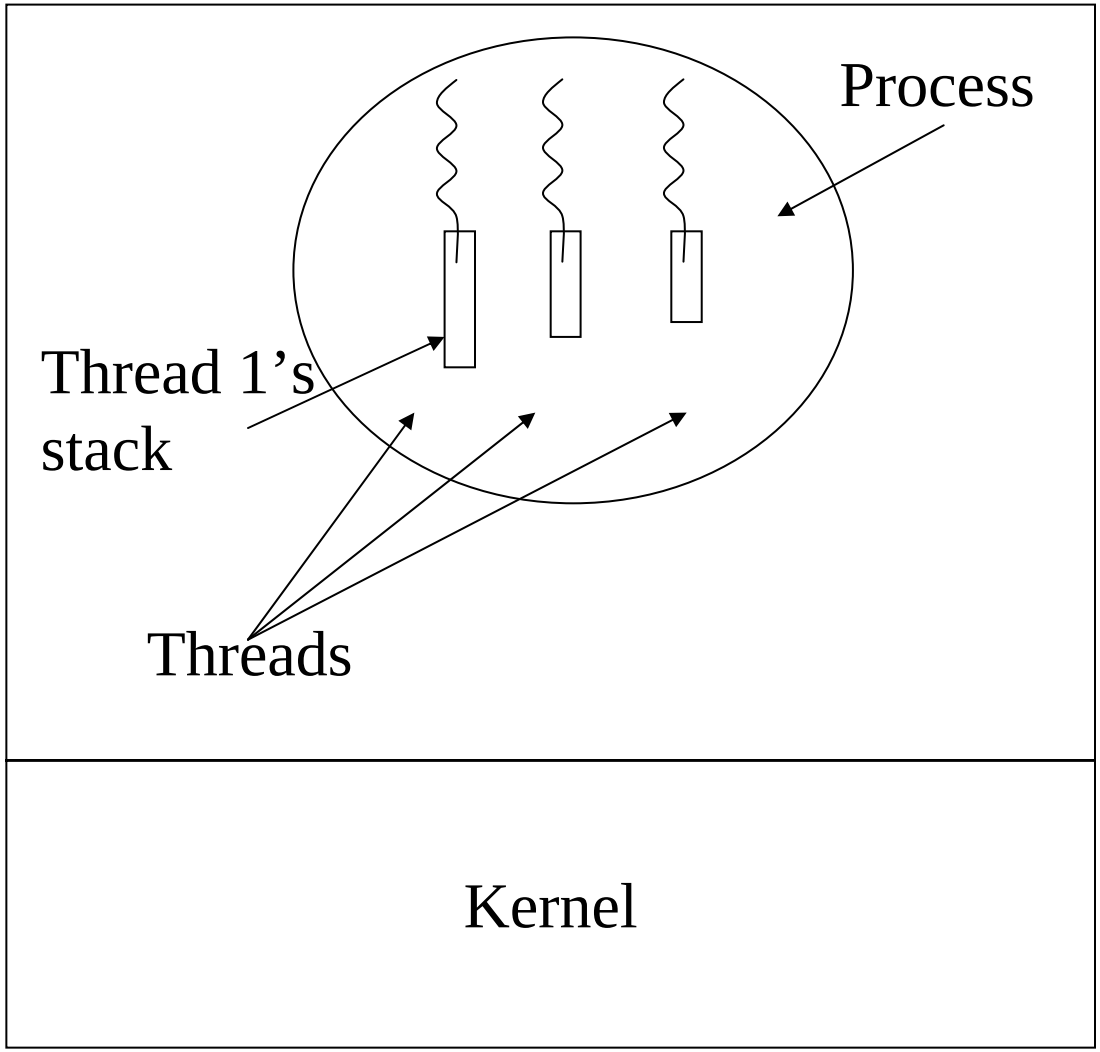
Use

- Processes are largely independent

Use

- Threads are part of the same “job” and are actively and closely cooperating

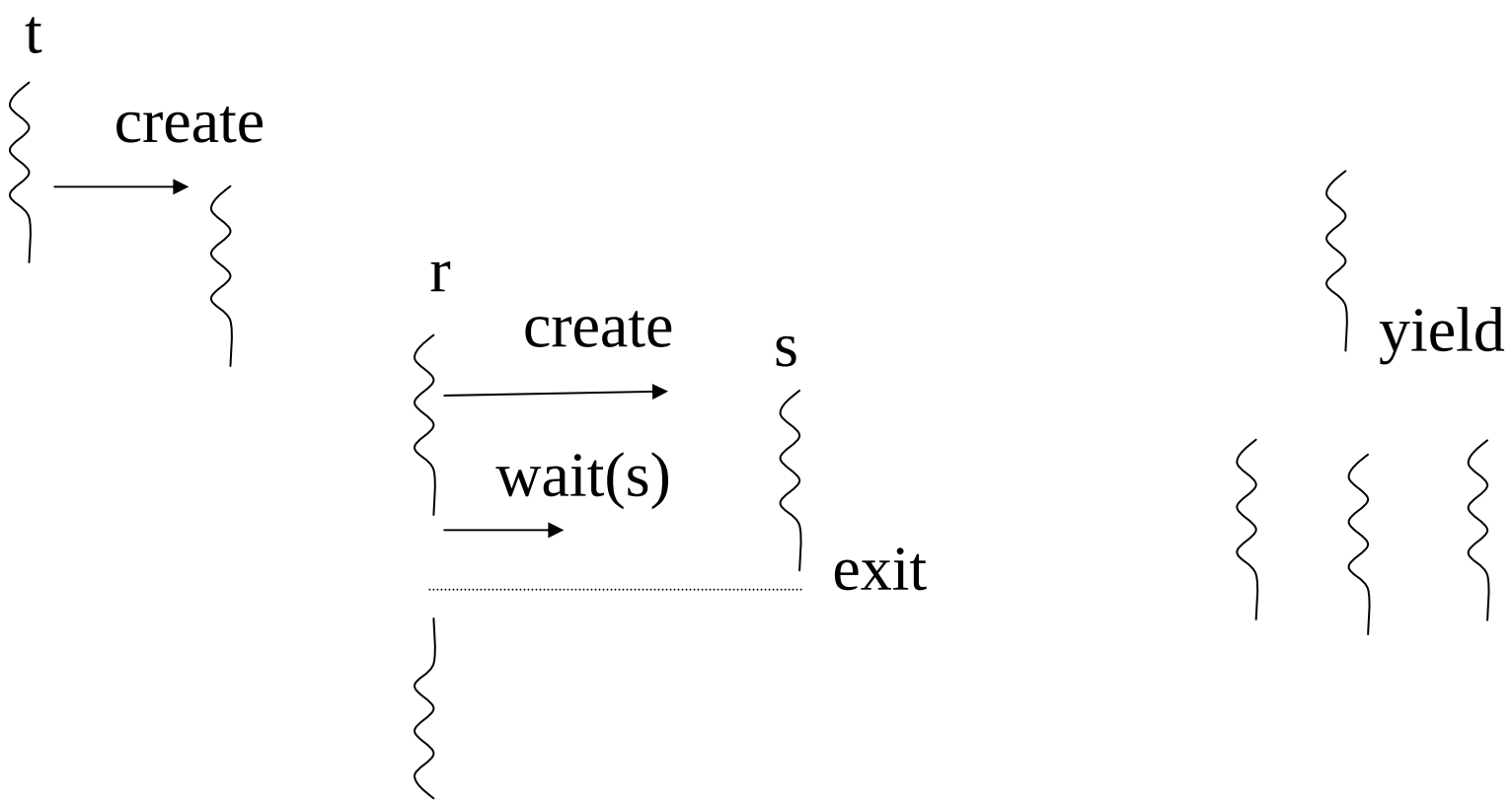
Thread Stack(s)



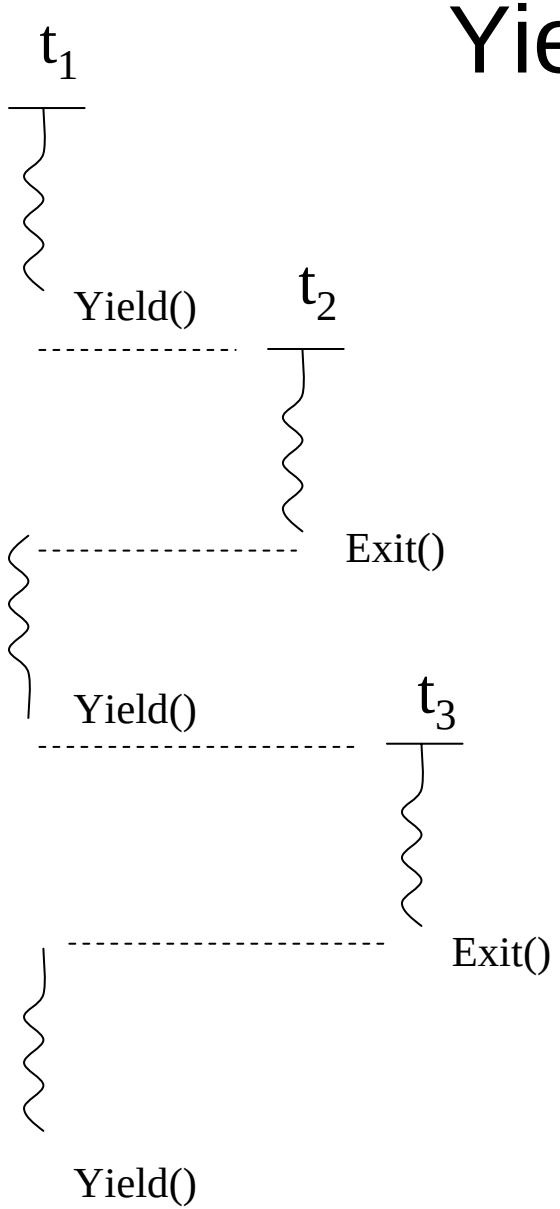
Common Thread Interface

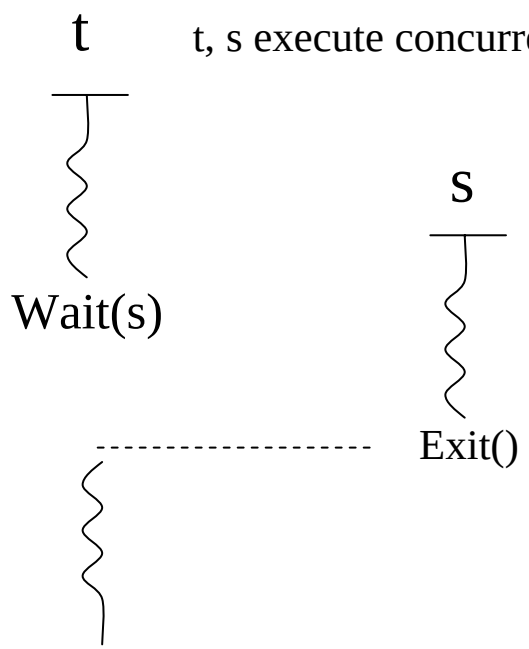
- `thread_create(…)`: creates a thread
- `thread_wait(…)`: waits for a specific thread to exit
- `thread_exit(…)`: terminates the calling thread
- `thread_yield(…)`: calling thread passes control on voluntarily to another thread

Thread Interface Illustrated

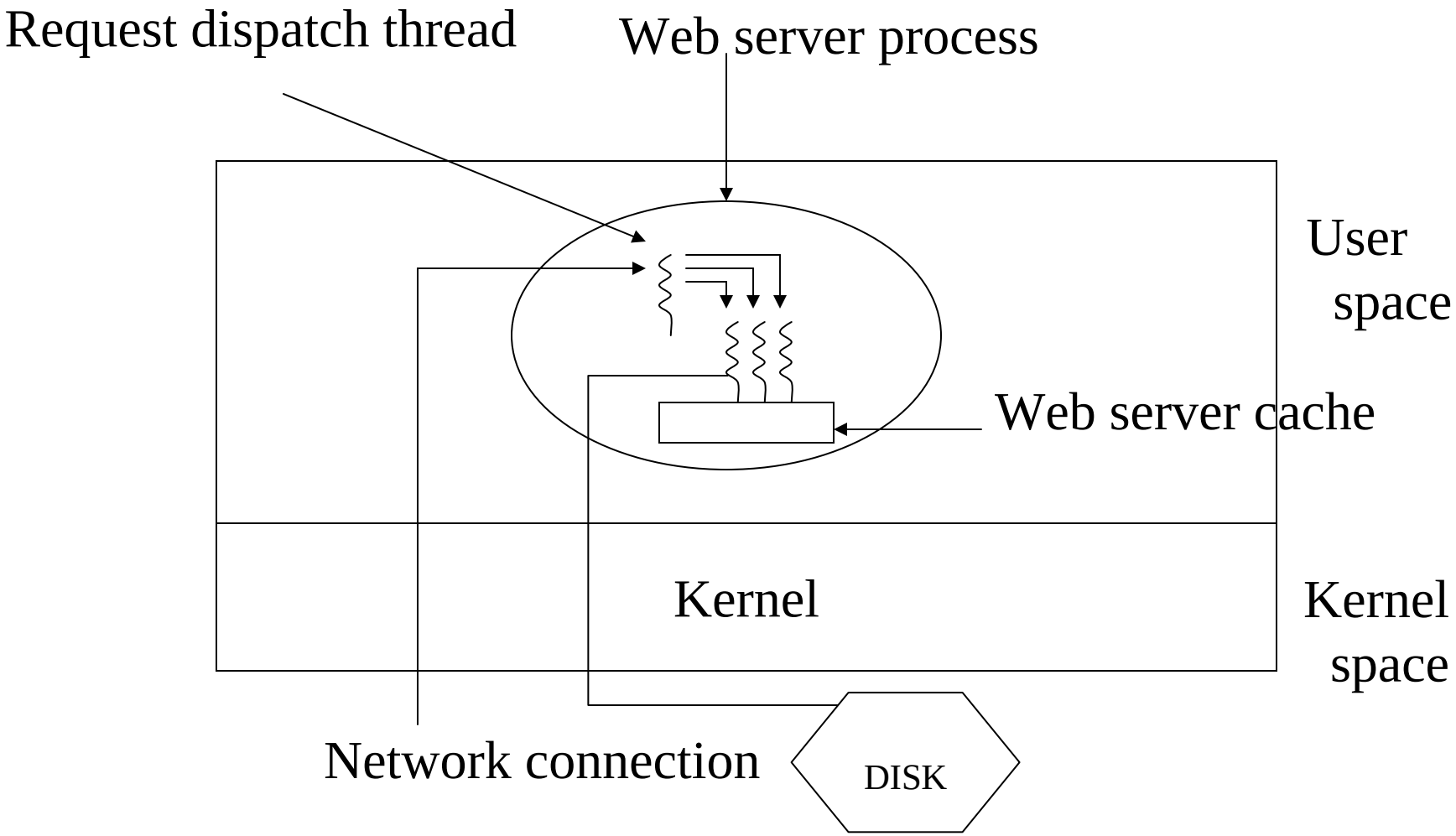


Yield vs Wait





Example: Multithreaded Web Server



Alternatives: Single Threaded Web Server Impl.

- Sequential processing of requests
- Finite state machine
  - event-driven prg. model
  - A.k.a. interrupt-driven model
- Processes instead of threads

Single Threaded Web Server Impl.

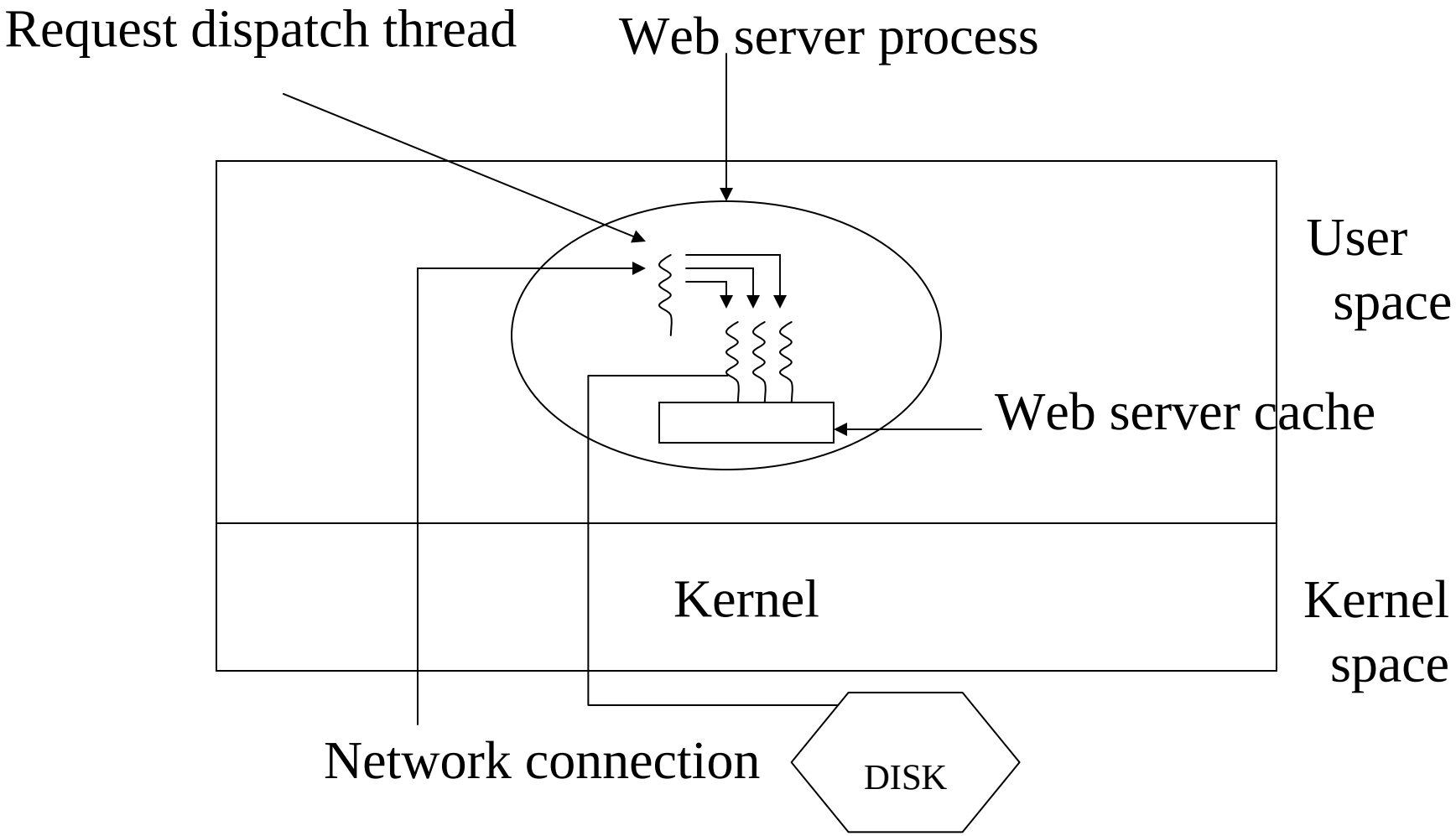
- Sequential processing of requests
  - Gets request, processes it, gets next
  - CPU idle while data is retrieved from disk
  - Poor performance
- Finite state machine (IRP/event-driven prg. model)
  - Use non-blocking system calls (read)
  - Record state of current request
  - Event: Get next request
  - Event: On reply from disk (signal/interrupt) process data read
  - Acceptable performance
  - Complicated to develop, debug …
- Processes instead of threads
  - Use of IPC facilities *et al*. for communication (e.g., many CGI and Web server implementations) (heavy weight solution)

FSM Algorithm

```
Loop //event loop
    get_nxt_event
    if event is a web-request
        …
        if not in web-cache
            update req-tbl
            non-blking-read()
    else if event is reply from disk
            update req-tbl
            send reply to client
//end loop
```

Non-blocking call, i.e., read and event loop
Execute concurrently.

Example: Multithreaded Web Server



Advantages of Threads in this Context

- **Application does not stall** when one of its operations blocks
- **Overhead for thread creation/destruction is low** - often much less than for processes
- **Simplification of programming model**
- **Performance gains** for machines with multiple CPUs

Model	**Characteristics**
Threads	Parallelism, blocking system calls
Single-threaded server	No parallelism, blocking system calls
FSM-based server	Parallelism, non-blocking s. calls, IRPs

Context Switch Times

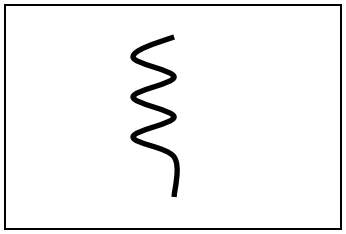
	Creation time (in microseconds)	Synchronization time using semaphores (in microseconds)
User thread	52	66
LWP	350	390
Process	1700	200

On SPARC station 2 from 1993, cf. **UNIX Internals the new foundation.** U. Vahalia.

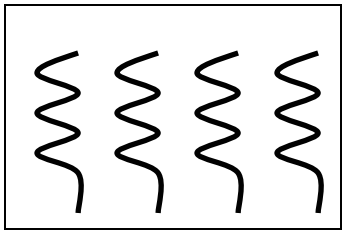
Benefits of Threads (generally speaking)

- **Increased concurrency** (i.e., interleave blocking operations; e.g., computation and I/O in one process)
- **Simplified programming model** (cf. IRP-driven programming)
- **Increased responsiveness** (balance of I/O and compute-bound threads)
- **Resource sharing** (i.e., leads to less memory being used)
- **Economy** (i.e., cheaper to create threads)
- Potential **performance gain**: Utilization of multi-CPU architectures

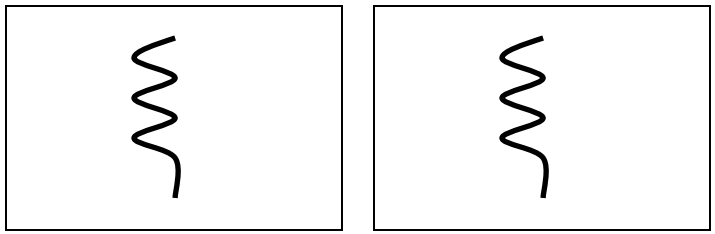
Threads and Processes



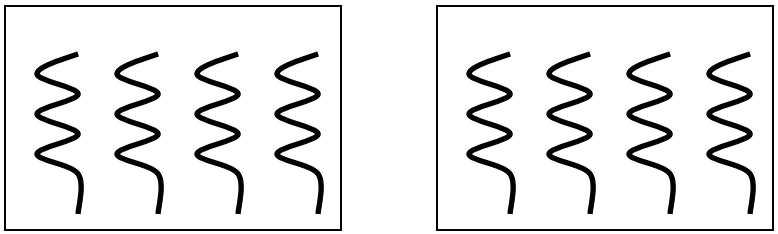
One process - one thread



One process – multiple thread



Multiple processes –
one thread per process



Multiple processes –
multiple threads per process

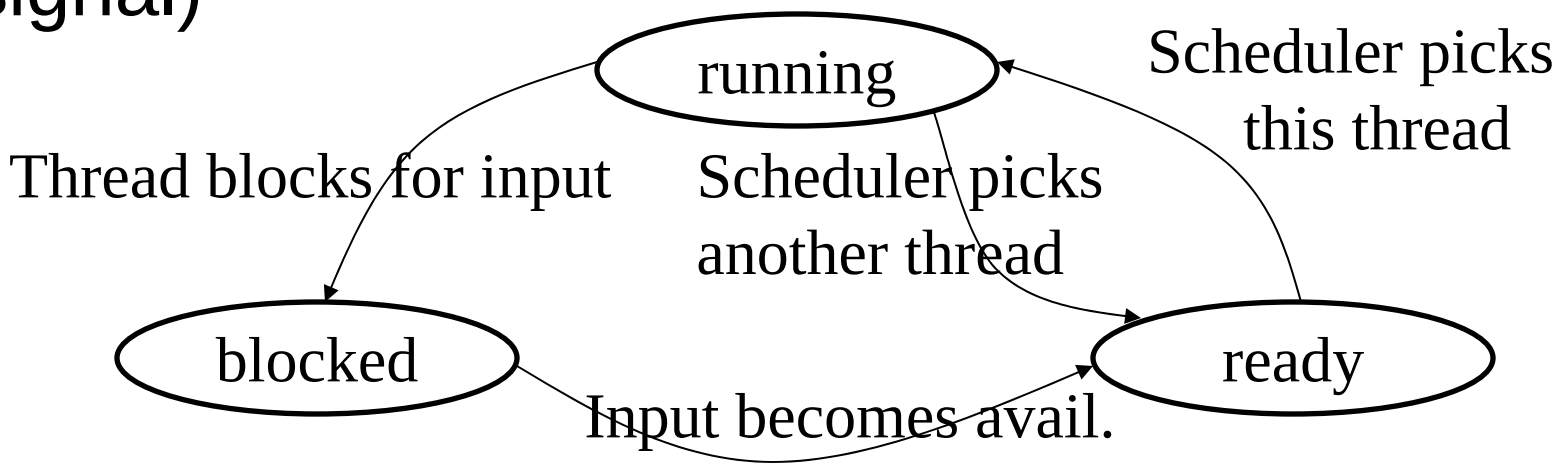
Threads and Processes

- Single process, single thread
  - MS DOS, old MAC OS
- Single process, multiple threads
- Multiple processes, single threads
  - Traditional UNIX
- Multiple processes, multiple threads
  - Modern UNIXes (Solaris *et al*.) and Windows 2000

Thread States

Three states (implementation dependent):

- **Running**: currently active and using CPU
- **Ready**: runable and waiting to be scheduled
- **Blocked**: waiting for an event to occur (I/O, signal)



Reasons for Leaving the Running State

- Thread **terminates**
  - `exit()` (system) call (termination)
  - `kill()` by another thread
- Thread **cannot continue** execution
  - **Blocked** waiting for I/O
- OS decides to **schedule another process**
  - Requires OS to be involved (e.g., system call or interrupt)
- Thread **voluntarily** gives another thread a chance
  - `yield()` system call

Implementation of Threads - Overview

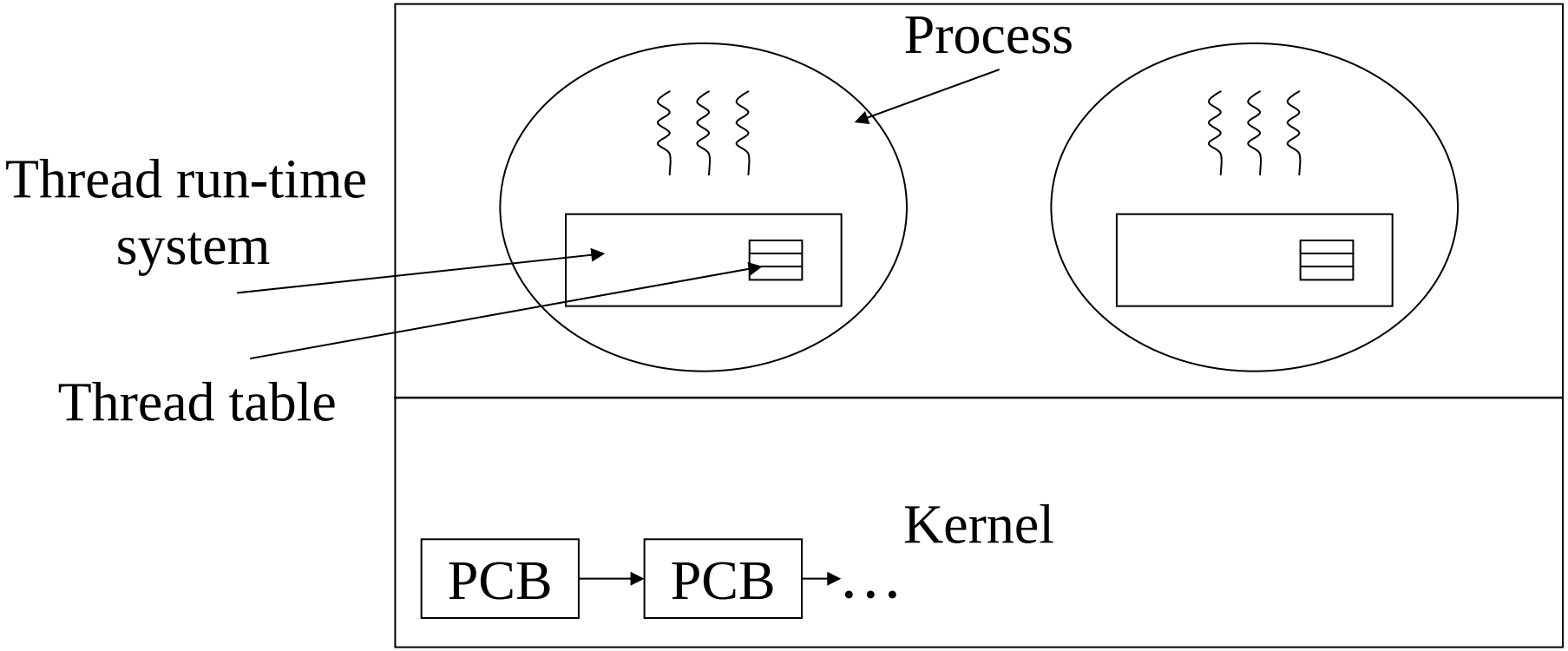
- In user space
- In kernel space
- Hybrids

User-level Threads

- Thread management done at the user-level, e.g., in a threads library (a.k.a. a *thread run-time system*)
- Kernel **knows nothing** about the existence of threads

- Examples
  - POSIX *Pthreads*
  - Mach *c-threads*
  - Solaris *threads*

User-level Threads



Calls into the run-time are like (local) procedure calls (no kernel mode switch, no trap, no context switch required.)

Advantages

- Can be supported on **operating systems that do not support threads inside the kernel**
- Thread context switch is much **faster** then process context switch
  - No kernel mode switch required
  - Only registers are saved/loaded
  - Thread may voluntarily yield (local call)
- Consequently, thread **scheduling is very efficient**
- May support per-process **customized scheduling** algorithms
- **Better scalability** (i.e., more threads, since no kernel access and data structures are required)

Issues

- Implementation of blocking system calls
  - A blocking system call will block the process
  - *The whole point was to allow for blocking calls in the first place*
- Could change sys. calls to be all non-blocking
  - Requires major rewrite of OS
  - *Counter objective of supporting user-level threads on many thread-unaware OSes*
- IRPs (e.g., page faults) may cause a context switch at “odd” times

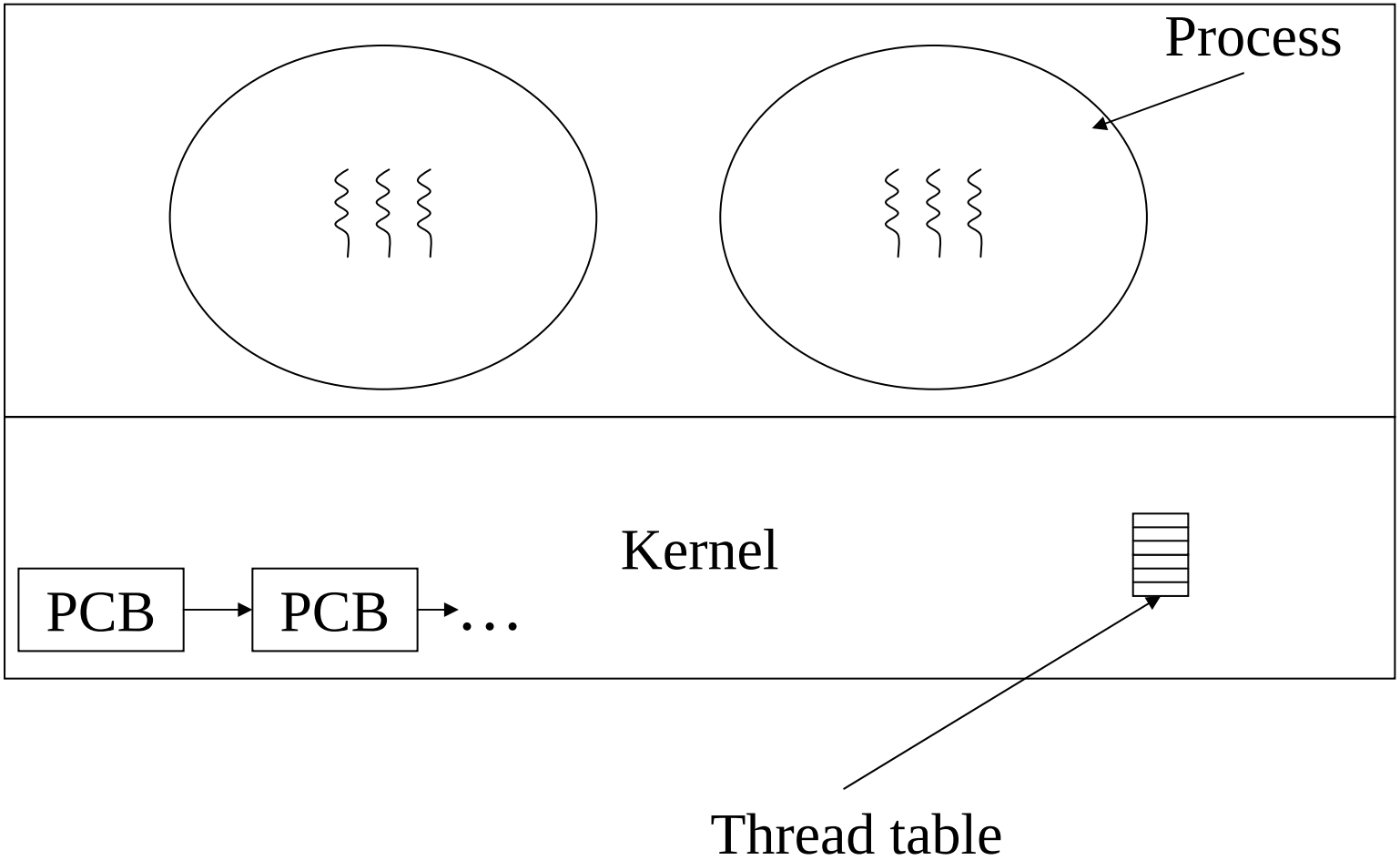
More Issues

- Threads may monopolize CPU (no clocks in process)
- Threads are introduced to interleave blocking calls with computation; however, once trapped, the OS can schedule further threads (hardly more work)
- No use for compute-bound tasks (unless multiple CPUs)

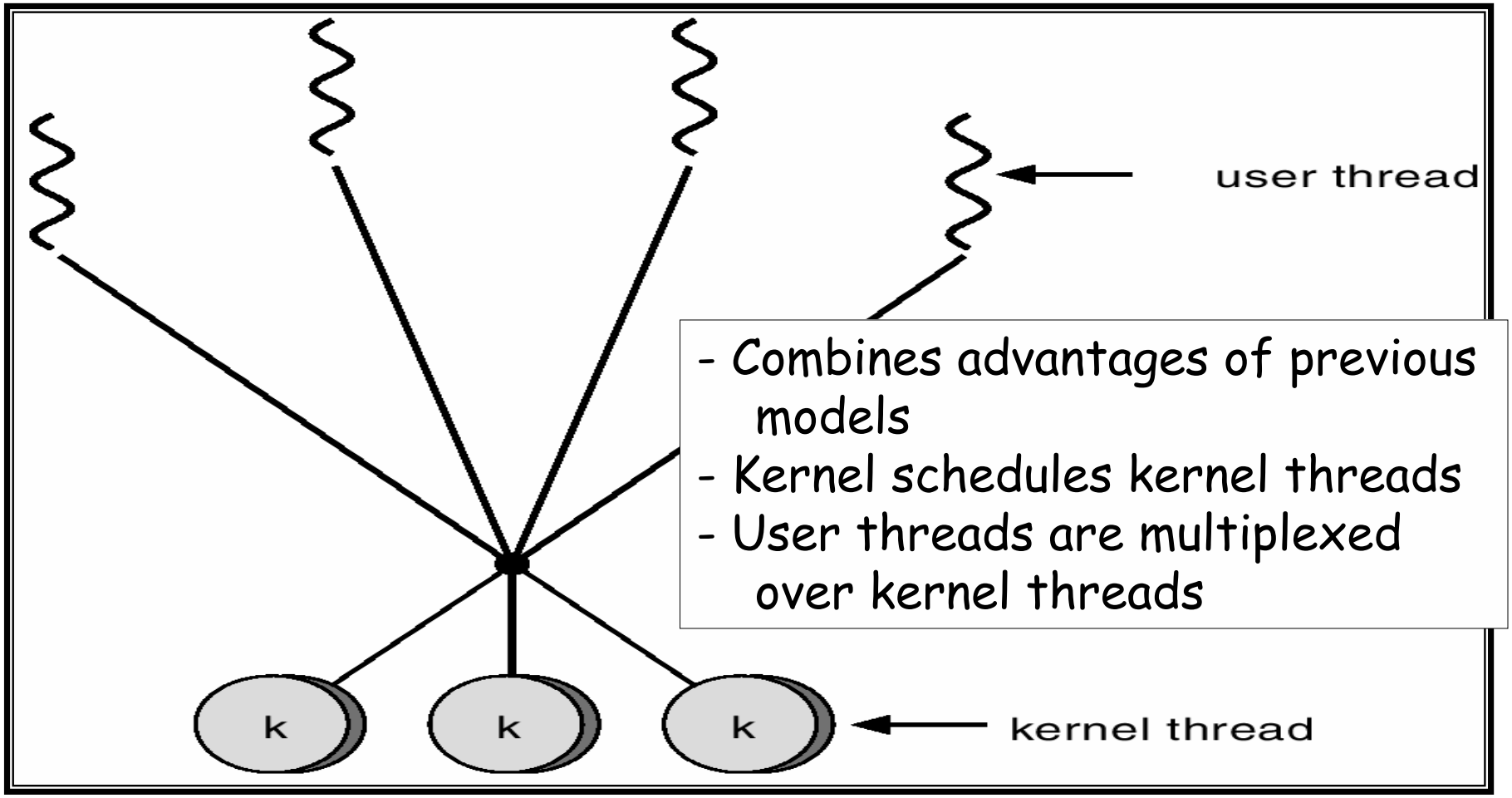
Kernel Threads

- Supported by the kernel
- Thread creation/destruction is done in the kernel; kernel data structures are manipulated
- Same thread management info as for user-level threads
- Mgmt. info is subset of process context
- Process context is also additionally managed
- Blocking calls yield scheduling decisions (same & diff. process)
- Thread recycling
- Considerable management cost and performance implications
- Examples: Windows 95/98/NT/2000, Solaris, Tru64 UNIX, Linux

Kernel Threads



Hybrid Model

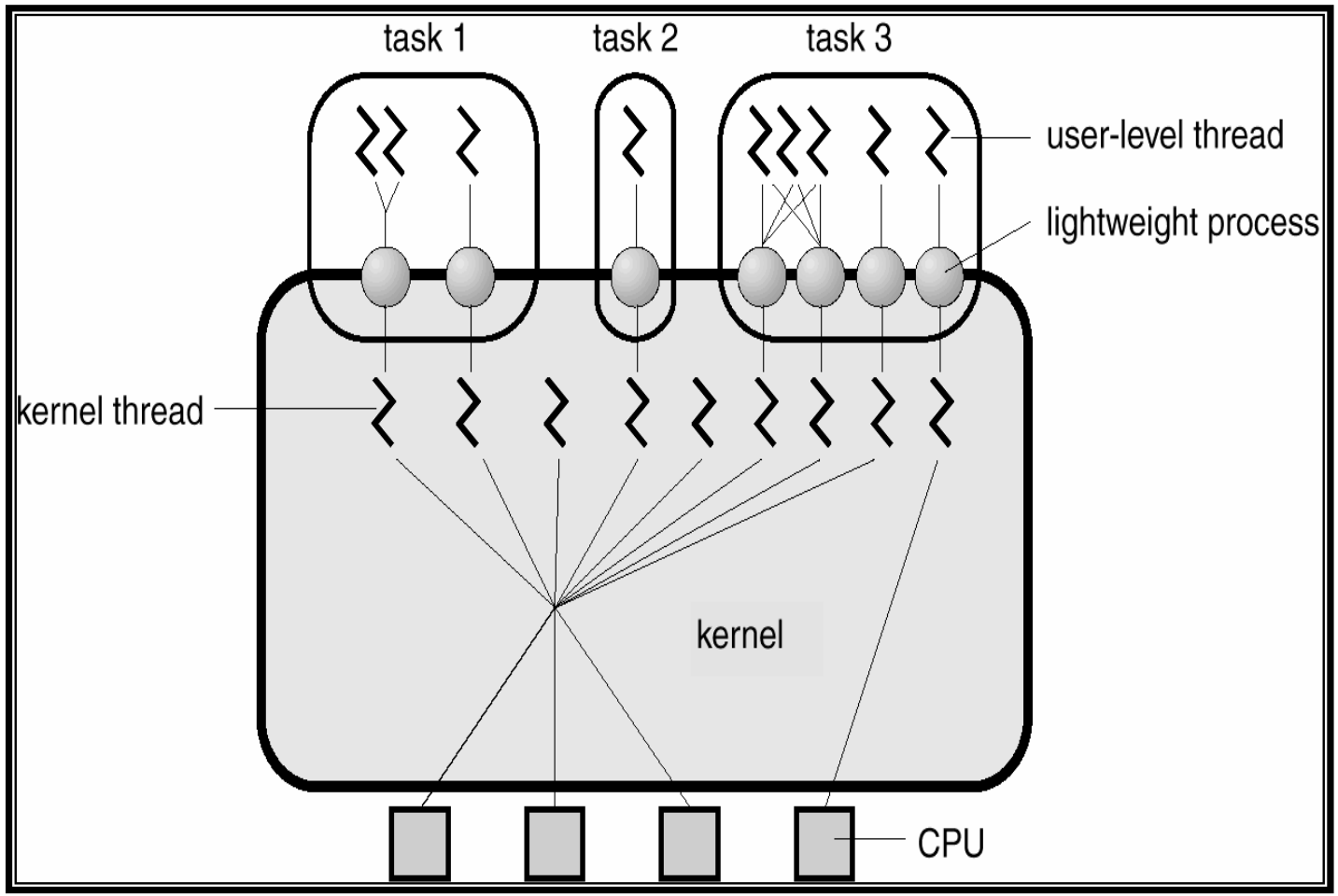


- Combines advantages of previous models
- Kernel schedules kernel threads
- User threads are multiplexed over kernel threads

Summary

- Limitations of traditional (single-threaded) process model
- Unit of resource ownership vs. unit of execution
- Multiple threads of execution
- Examples of multithreaded applications
- Alternatives to multithreading (i.e., FSM-based)
- Benefits of threads
- Use of threads vs. use of processes
- Thread implementation, i.e., user vs. kernel threads and hybrids

Example: Solaris 2 Threads



Solaris Process