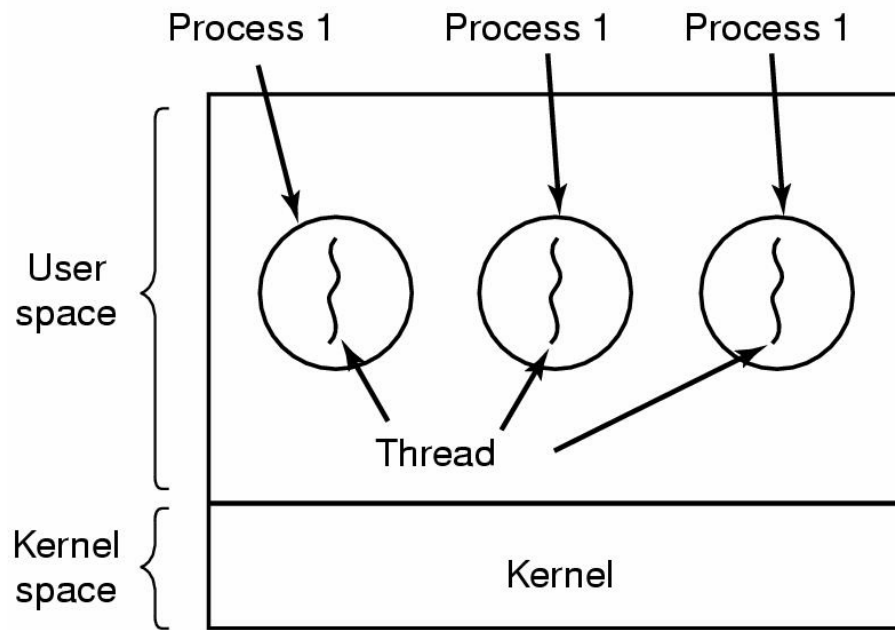


Betriebssysteme

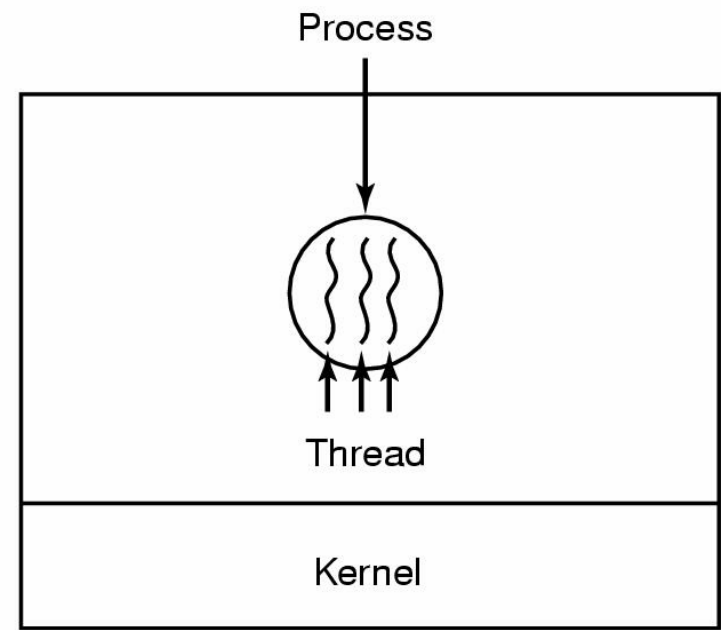
Threads

Prof. Dr.-Ing. Torben Weis
Universität Duisburg-Essen

Prozesse & Threads



(a)



(b)

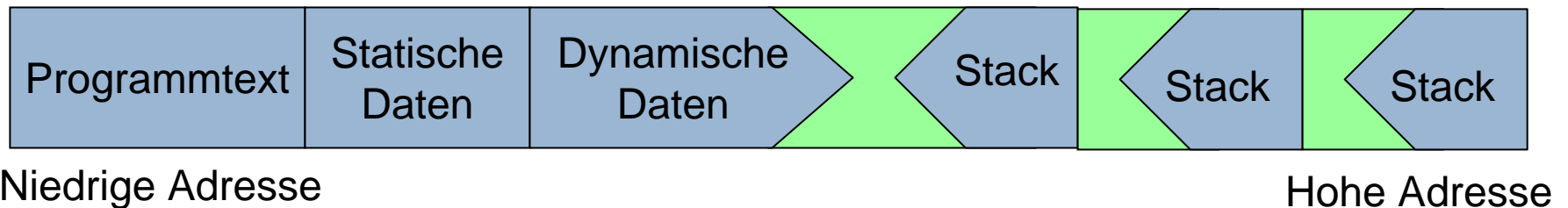
Prozesse & Threads

Elemente pro Prozess

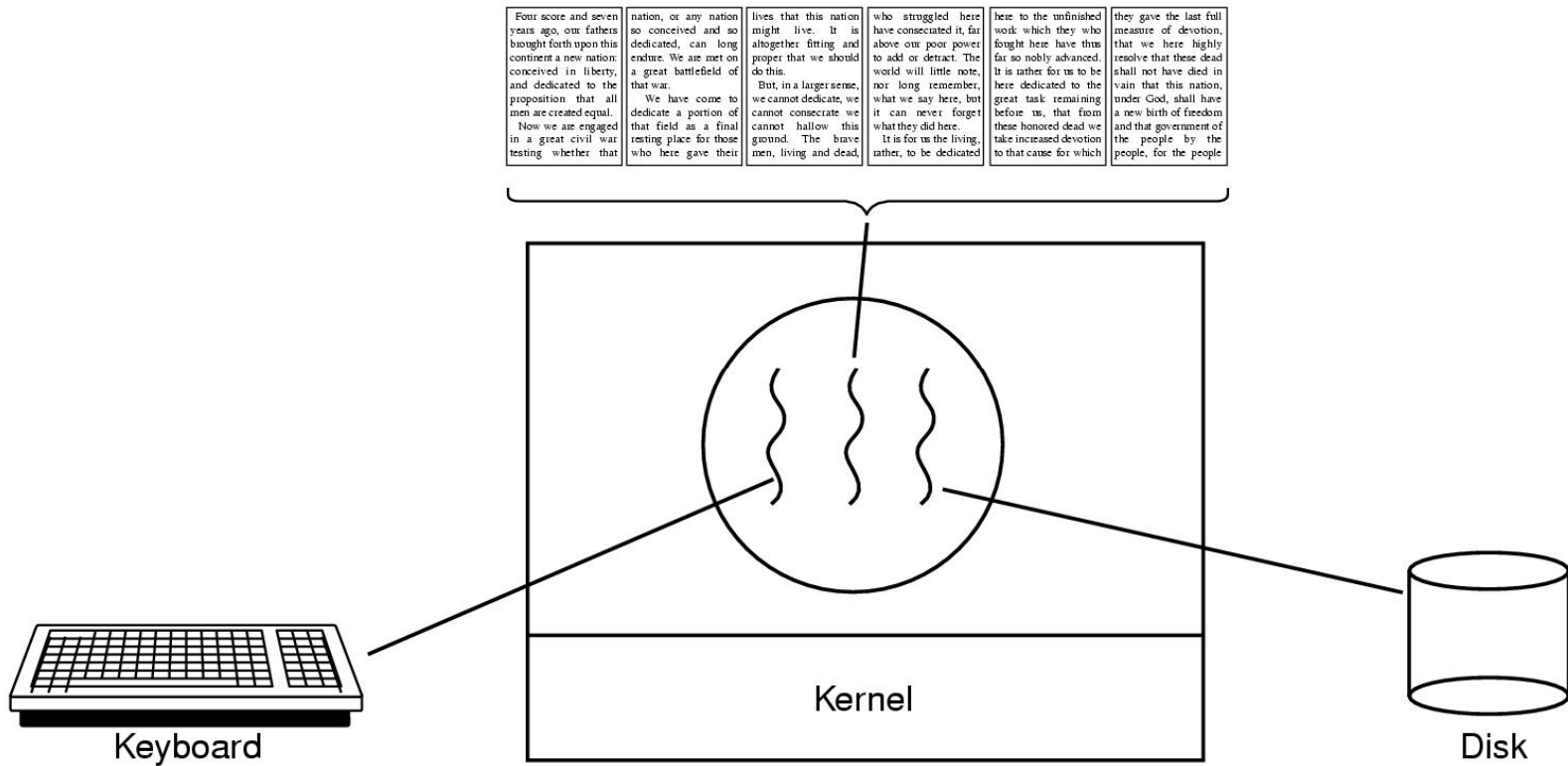
- Adressraum
- Geöffnete Dateien
- Kindprozesse
- Signale
- Accounting
- Security-Policies

Elemente pro Thread

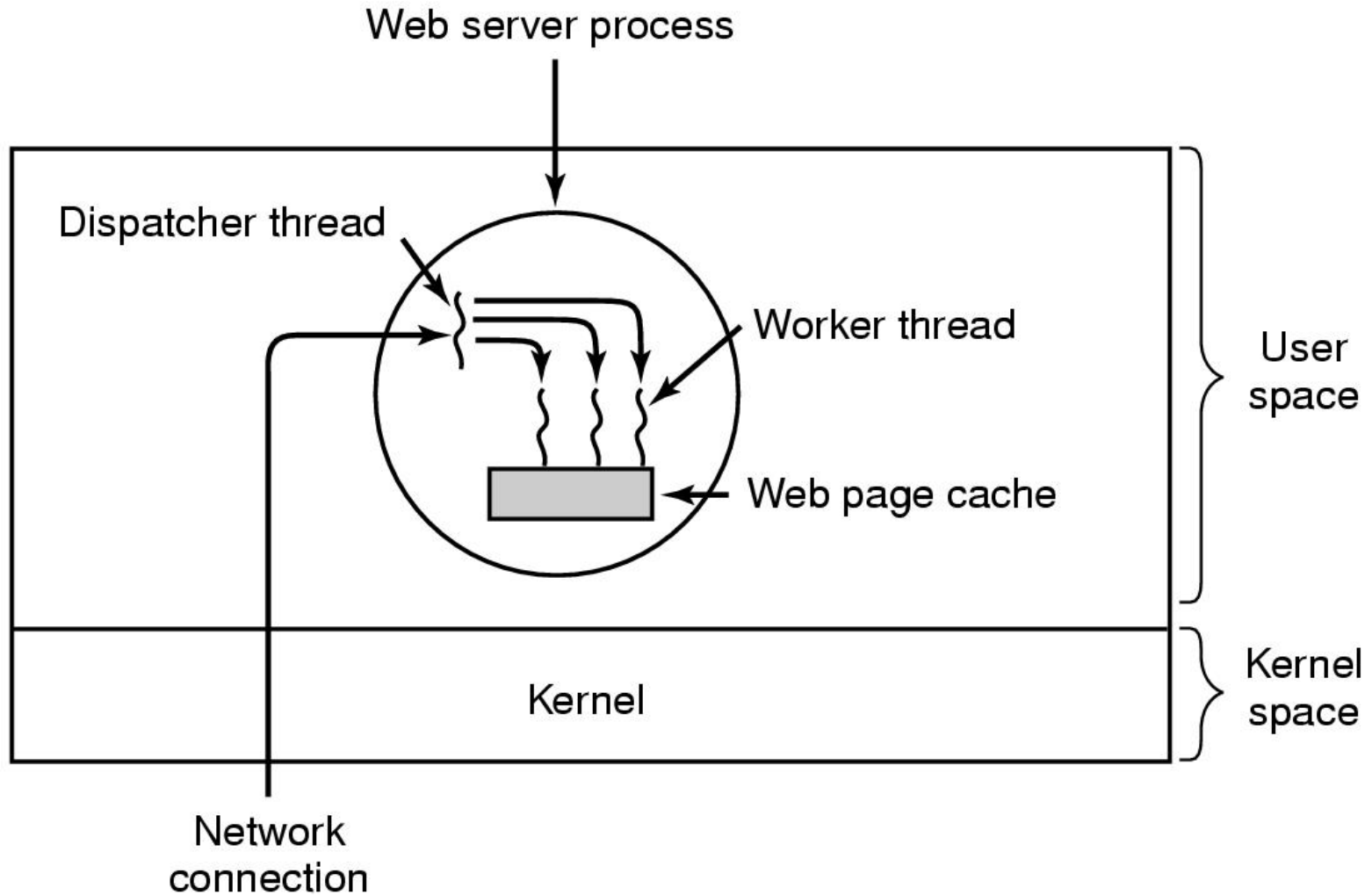
- Befehlszähler (PC)
- Register
- Keller (SP + Speicher)
- Zustand



Beispiel: Textverarbeitung mit 3 Threads



Beispiel: Multithreaded Web-Server



Dispatcher und Worker Threads

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

(a)

```
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);  
}
```

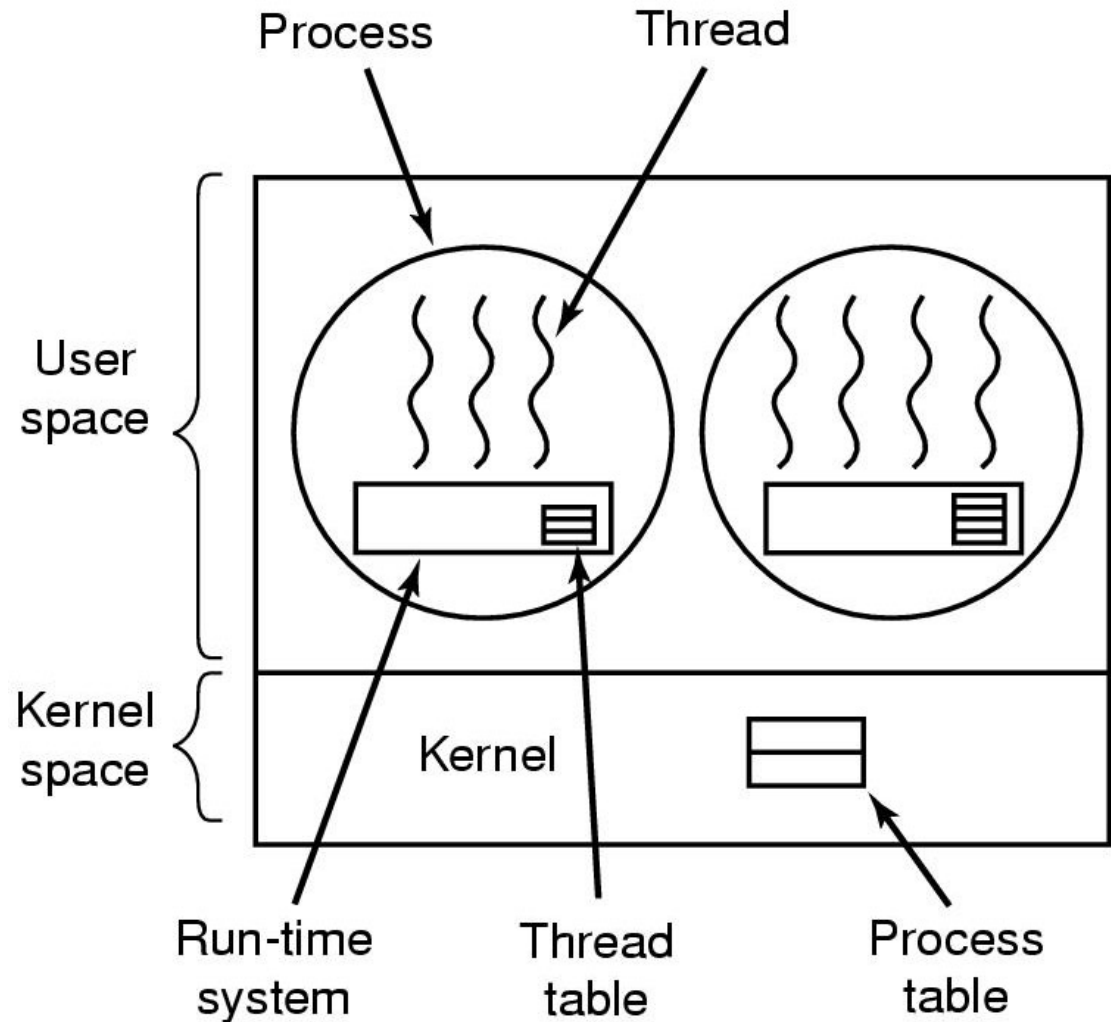
(b)

- (a) Der Dispatcher Thread akzeptiert Anfragen und leitet sie an andere Threads weiter
- Blockiert bei „get_next_request“
 - Blockiert nicht bei „handoff_work“
- (b) Die Worker Threads beantworten Anfragen
- Können jederzeit blockieren

Konstruktion eines Servers

- Multi-Threaded
 - Parallelität
 - Blockierende Systemaufrufe
- Single-Threaded
 - Keine Parallelität
 - Blockierende Systemaufrufe
- Endliche Automaten
 - Parallelität
 - Nicht-blockierende Systemaufrufe oder Interrupts

User-Level Threads



User-Level Threads

- Vorteil 1: Scheinbar schneller
 - Thread-Wechsel erfordert keinen Kernel-Trap
 - Daher weniger Overhead bei Thread-Wechsel
 - Schont Betriebssystem-Ressourcen wenn sehr viele Threads verwendet werden
- Vorteil 2: Austauschbare Scheduler
 - Jeder Prozess kann einen anderen Thread-Scheduler einsetzen
- Vorteil 3: Kontrolle über Thread-Wechsel
 - Beispiel: Thread-Wechsel während der Garbage-Collection kann einfach verhindert werden

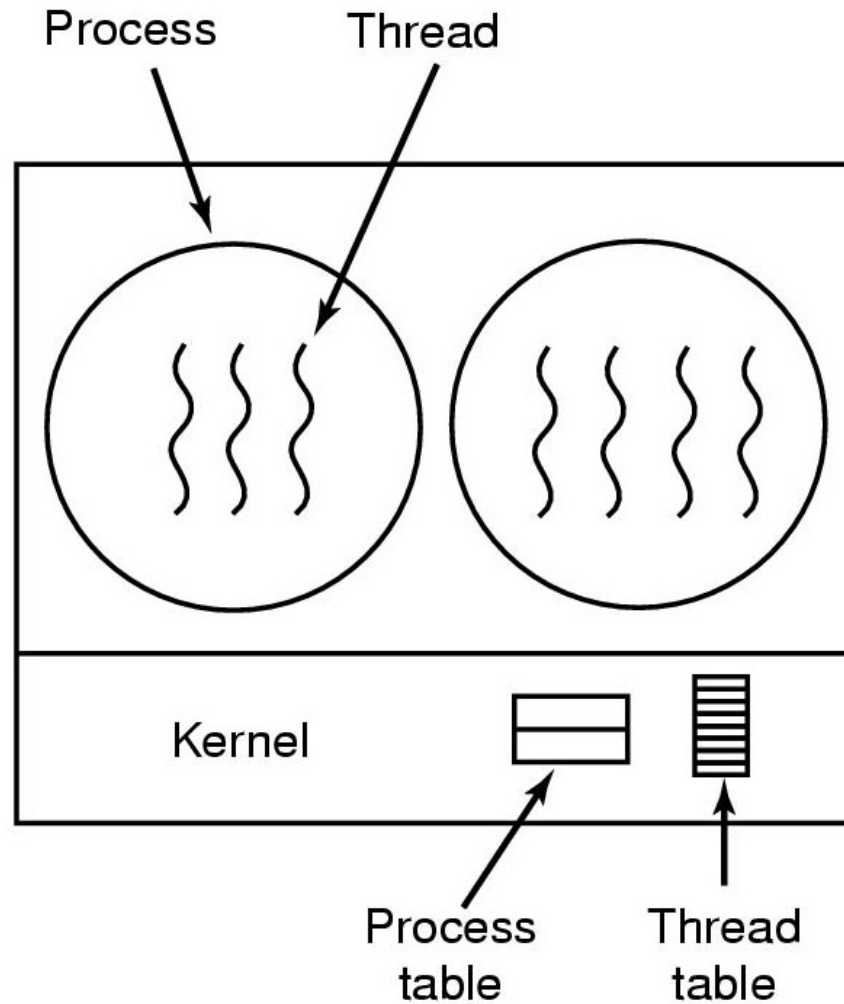
User-Level Threads

- Thread-Wechsel
 - Aufruf einer blockierenden Systemfunktion
 - Aufruf von `thread_wait` oder `thread_yield`
d.h. kooperatives multi-threading
 - Kein Interrupt zur Unterbrechung !!!
- Blockierende Systemfunktionen
 - Der Prozess darf nicht blockieren
 - Lösung A
 - Abbildung auf nicht-blockierende Funktionen
 - Lösung B
 - Vorher Abtesten, ob die Funktion blockieren wird
 - Beispiel: `select` vor `read` oder `write` aufrufen

User-Level Threads

- **Nachteil 1: Zusätzlicher Code**
 - Wrapper um jede Systemfunktion notwendig
- **Nachteil 2: Blockierende Aufrufe**
 - Blockierende Systemaufrufe sind teuer
 - Beispiel: `select + read = 2 Kernel-Traps`
- **Nachteil 3: Kooperatives multi-threading**
 - Threads müssen freiwillig die Kontrolle abgeben
- **Nachteil 4: Seitenfehler**
 - Bei Seitenfehlern wird der ganze Prozess suspendiert, obwohl man nur hätte den Thread wechseln müssen

Kernel-Level Threads



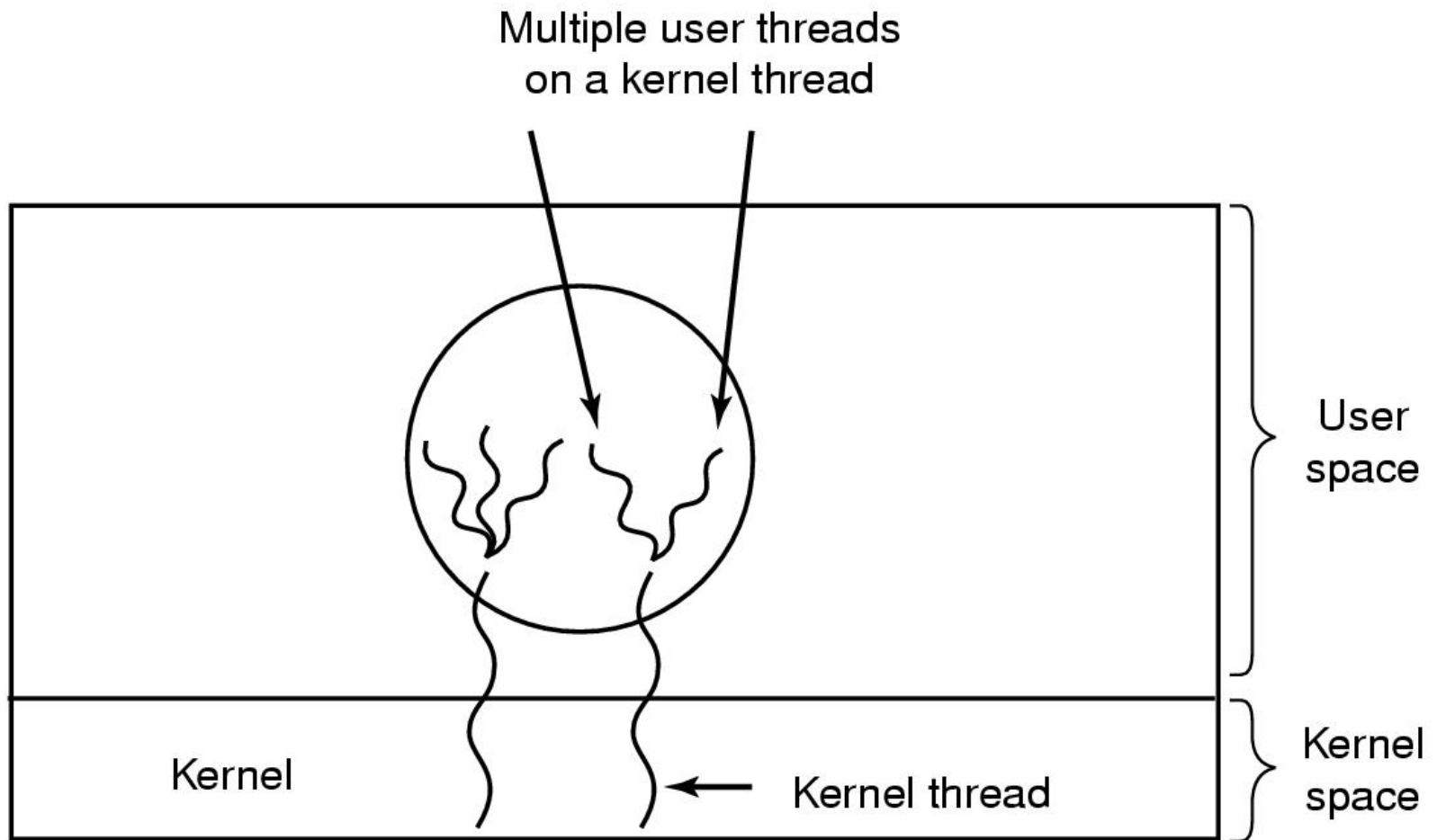
Kernel-Level Threads

- Vorteile
 - Besser bei blockierenden Systemaufrufen
 - Kein Problem mit Seitenfehlern
 - Preemptives Multi-Threading
- Nachteile
 - Jede Thread Operation ist teuer, weil ein Systemaufruf notwendig wird
 - Besonders Erzeugen/Zerstören bedeutet Änderungen in Kernel-Datenstrukturen

Thread-Pool

- Threads kann man auf Vorrat halten
 - Das spart Kosten für Erzeugen/Zerstören
 - Zuteilungen aus dem Thread-Pool erfolgen sehr schnell im User Space, d.h. ohne Systemaufruf
- Besonders sinnvoll um Worker-Threads zu implementieren
 - Worker-Threads laufen eventuell nur sehr kurz
 - Dadurch werden die Kosten für Erzeugen/Zerstören deutlich spürbar
- Nicht sinnvoll bei User-Level Threads
 - Hier ist das Erzeugen/Zerstören kein Problem

Hybride Implementierung



Threads und globale Variablen

