Vorlesung 6

Single- und Multitasking

Peter B. Ladkin
ladkin@rvs.uni-bielefeld.de

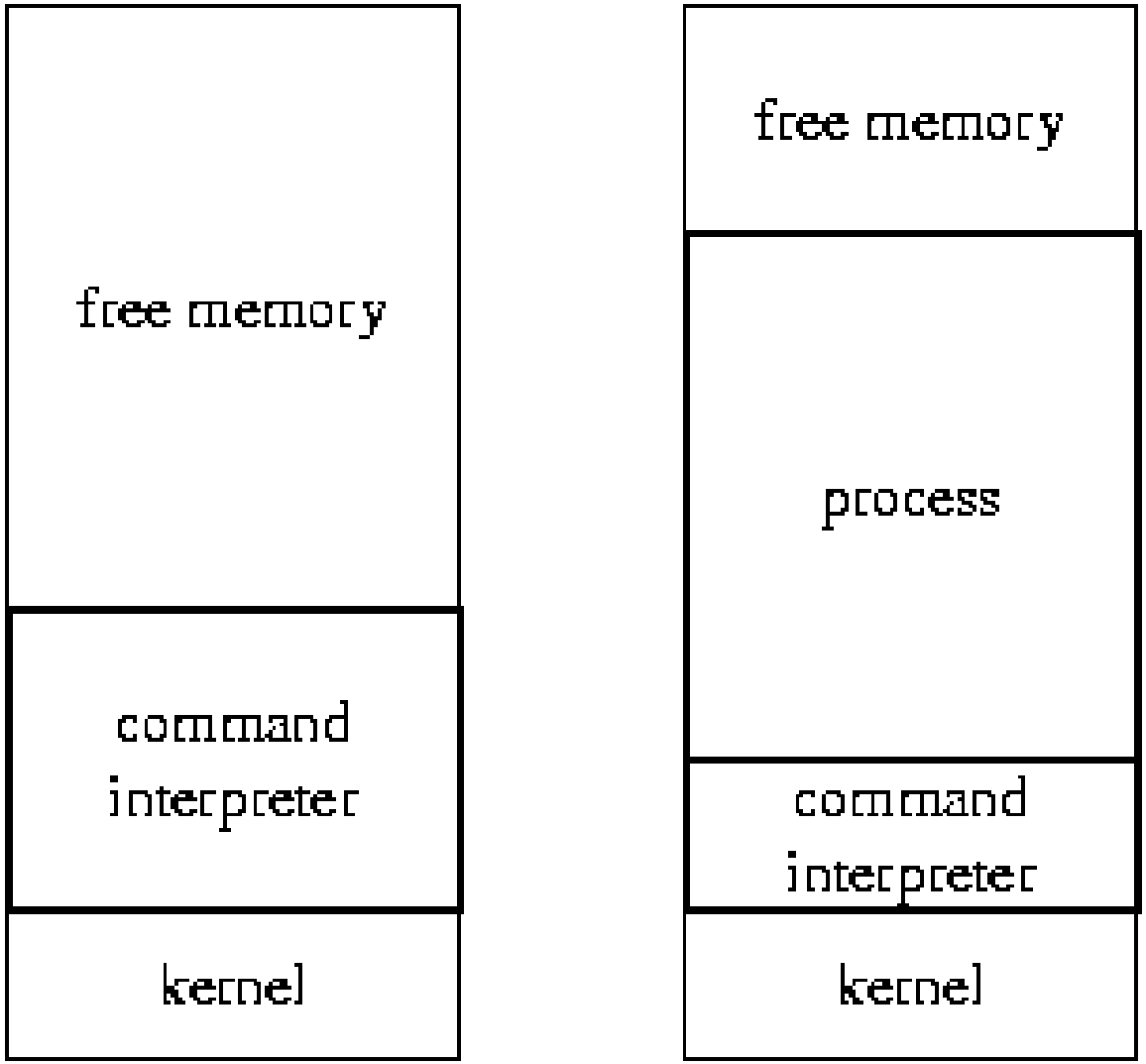
Single Tasking

- Command Interpreter (ComInt) läuft
- wartet auf Tastatur-Eingabe
- "liest" (parst) die Eingabe (für Prog-Name)
- Macht "Lookup" von <Prog-Start-Adresse>
- Führt "JMP <Prog-Start-Adresse>" aus
- Jedes Programm muss
  "JMP <ComInt-Adresse>" am Ende ausführen

Single Tasking

- Einfach zu programmieren
- braucht keine Clock-Interrupts
- kann Busy-Wait für I/O machen (warten muss er sowieso)
- Wenn ein Program fehl schlägt, hängt der Rechner
- Bekanntes Beispiel: MS-DOS

Single Tasking



Single-Tasking: MS-DOS

- Basis für Windows-3.x, Windows-95, 98 usw.
- Die "persönlichen" Versionen
- Aber nicht Windows-NT
- MS-DOS ist 16-Bit (im IBM-XT mit 8-Bit Datenübertragung auf Grund des Busses der CPU)

Multitasking

- Programme laufen "gleichzeitig"
- Ein Scheduler-Programm (BS) verteilt die Prozessor-Zeit auf die laufenden Programme
- Ein Programm kann also zweimal während der gleichen Zeit ausgeführt werden
- Wir sprechen nicht mehr von Programmen, sondern besser von "Prozessen"

Zeit-Ablauf

- Clock Interrupts
- Time-Slicing über den Clock Interrupt Handler
- Handler
  - **loop**

```
    if i > 0 then i <- (i-1)
             else
                  Store(State);
                  i <- 1000
                  Scheduler;

endloop
```

Prozesse

- Ein Prozess ist ein "Programm im Lauf"
- Besteht aus "aktuellem Zustand" (Current State)
- Current State beinhaltet:
  - Werte vom PC und allen Registern
  - Die Werte der Programm-Variablen
  - Ist das Programm *Ready*, *Waiting*, *Running*, *Terminated*?

Process Table

- Die Prozess-Zustände werden in einer Tabelle gespeichert: Die “Process Table”
- Die Process Table beinhaltet den Current State (PC und Register-Werte) plus Status (*Ready*, *Waiting*, *Running*, *Terminated*)
- Wenn *Running*, Current State nicht aktuell
- Wenn *Terminated*, Current State nicht aktuell

Queues

- *Ready* Prozesse müssen zugeordnet werden
- Wenn ein I/O Interrupt signalisiert, dass I/O fertig ist, wird der Handler den entsprechenden Prozess von *Waiting* zu *Ready* umstellen
- Wenn ein Prozess nur wegen Zeitablaufs unterbrochen wird, wird er als *Ready* gehalten
- Sonst wegen I/O als *Waiting*

Ausswappen

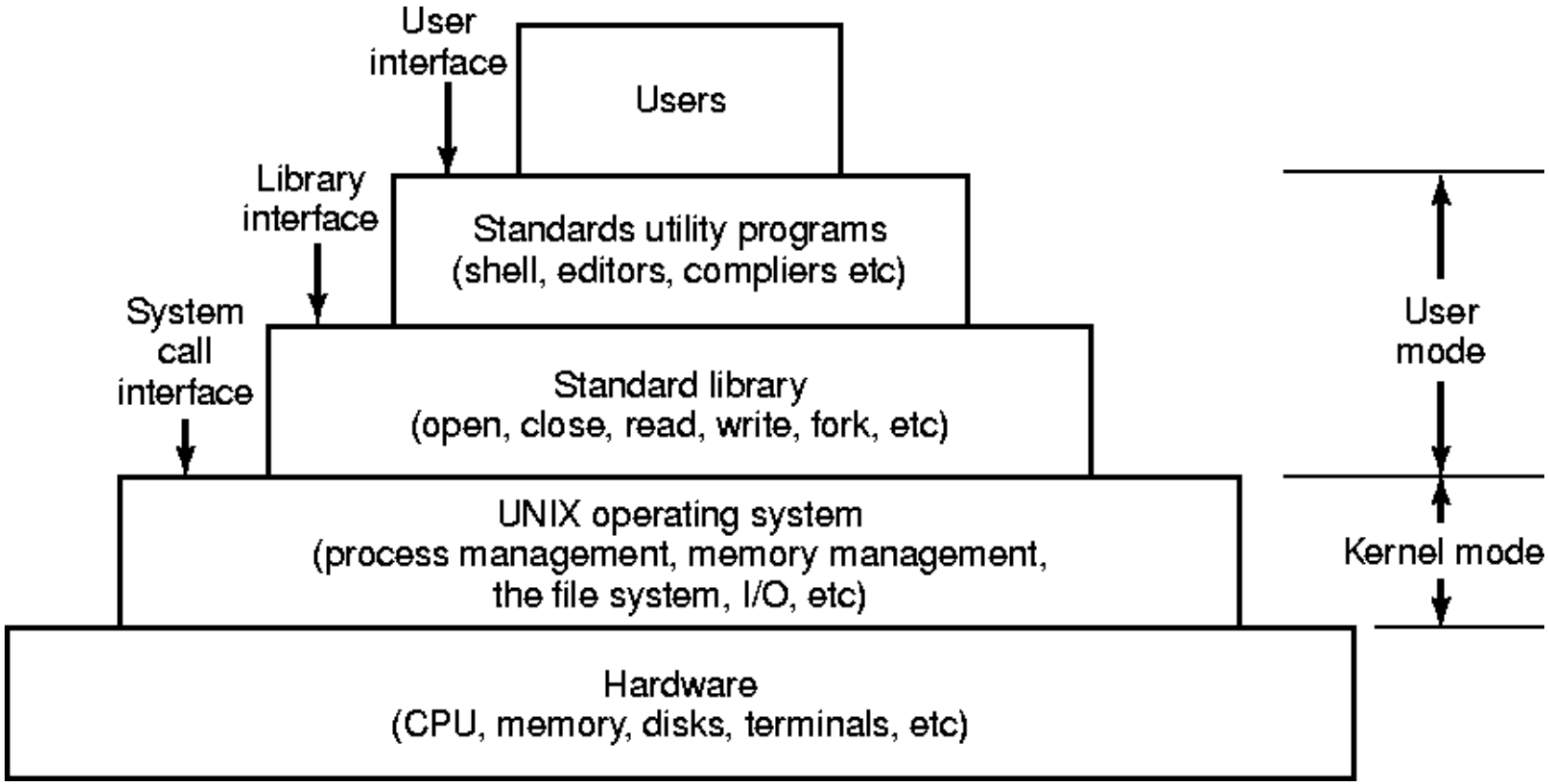
- Scheduler
  - **Case:**

```
timeout:
  Store(CurrentState,ProcTab.ReadyQ);
  Load(Head(ProcTab.ReadyQ)
I/O wait:
  Store(CurrentState, ProcTab.WaitingQ)
  Load(Head(Processtable.ReadyQueue)
```

Queues

- Verwaltung von Ready-Queue ist nicht unbedingt einfach
- Ready-Prozesse könnten klassifiziert werden
  - Auf der nächsten Folie:
    Eine 5-fache Klassifizierung

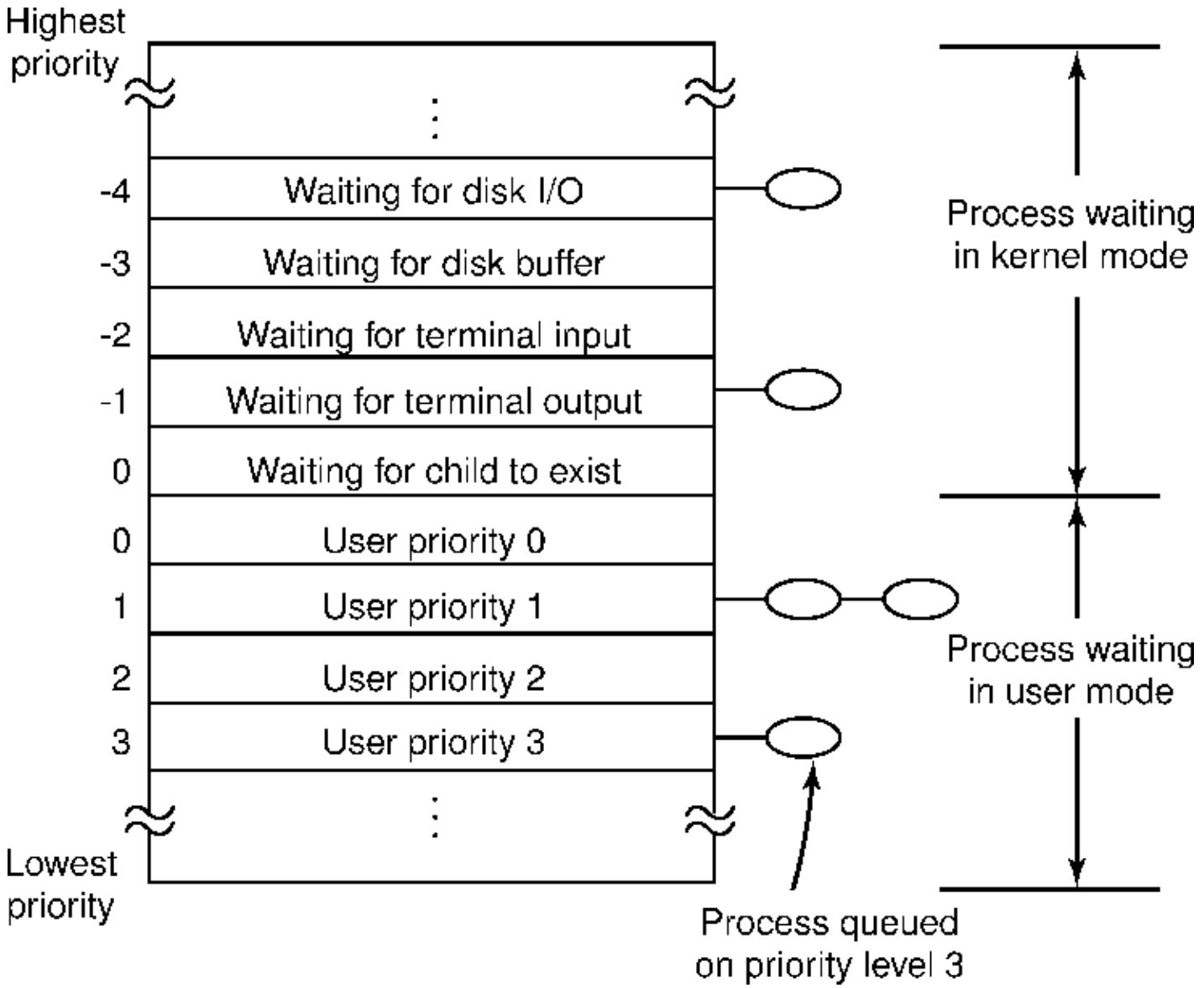
Prozess-Klassifizierung



Prozess-Klassifizierung

- Auf der nächsten Folie:
  Wie die Prozess-Tabelle mit Prioritäten, sowie auch Begründungen für Waiting-Status (auch eine Art Klassifizierung) aussieht

Prozess-Tabelle

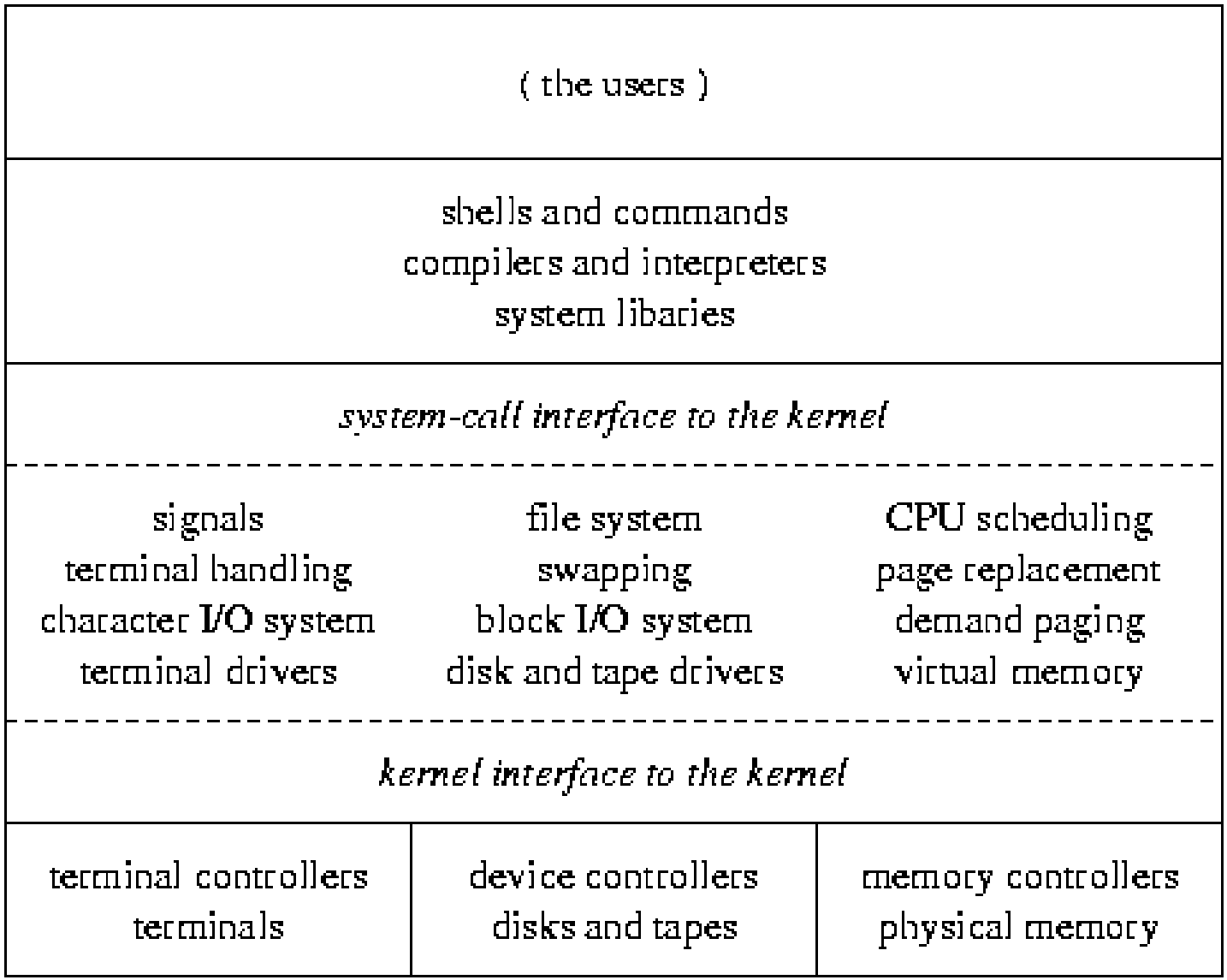


Prozess-Klassifizierung

- Auf der nächsten Folie:

  Eine feinere Klassifizierung
  der Betriebssystem-Software

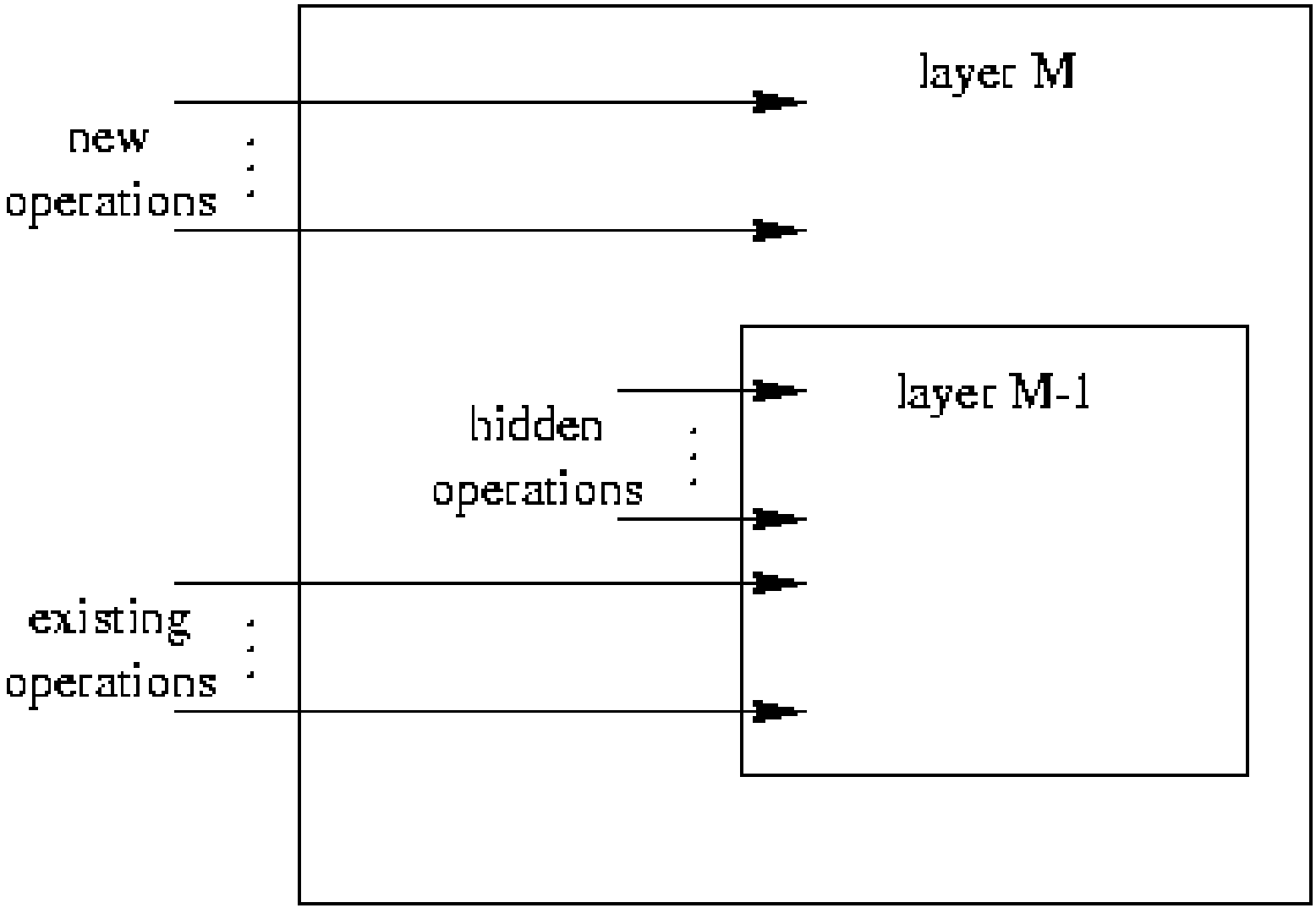
Prozess-Klassifizierung



Prozess-Klassifizierung

- User benutzen Shells, Compiler, usw.
- Shells, Compiler usw. benutzen Signale, Character I/O System, File-System, usw.
- Signale, Character I/O, File-System benutzen Terminal-Kontroller, Device-Kontroller, Speicher-Management
- Hierarchie von virtuellen Maschinen
  (wie in Vorlesung 3 eingeführt)

Layers


layer M
new operations
hidden operations
layer M-1
existing operations

Ready-Prozess-Auswahl

- 3 Ready-Prozesse: A, B, C
- Prioritäten: A:1, B:2, C:3
- 3 ist höchste
- Prozess mit höchster Priorität wird nach Zeitablauf als Running ausgewählt
- Konsequenz: A läuft nie - **Starvation** oder **Indefinite Blocking**

Probleme des Scheduling

- Wir haben **Starvation** gesehen
- Ausserdem gibt es den **Deadlock** -
  - Zwei oder mehr Prozesse warten gegenseitig aufeinander (d.h. der eine wartet, bis der andere was tut, und der andere wartet, bis der erste was tut)
- Die zwei größten Probleme des Scheduling
- Scheduling ist nicht trivial

Beispiel: Scheduling Policy I

- Jeder *Ready* Prozess wird gestartet und läuft bis zum Ende
- Single-Tasking

Beispiel: Scheduling Policy II

- Jeder Prozess läuft für eine bestimmte Zeit, wird dann ausgeswappt und wartet als *Ready* Prozess
- Multitasking

Scheduling Policy Auswahl

- Wann ist welche Policy sinnvoll?
- Policy I
  - Einfache Systeme mit beschränkten Ressourcen
  - Ein-Benutzer Systeme
  - Echtzeit Systeme, in denen jeder Prozess und dessen genauer Zeitlauf bekannt, vertraut, wichtig und Hard-Deadlined ist
  - Naturwissenschaftliche Supercomputer

Scheduling Policy Auswahl

- Wann ist Policy II sinnvoll?
  - General-Purpose Computer
  - Business Info-Systeme
  - WWW- und Internet-Server
  - Transaction-Processing
    - Bank ATM
    - Flugkarte/Bahnfahrkarte Reservierungssystem
    - Allgemeine Informationssysteme - Amedeo

Policy II: Subpolicies

- First Come First Served
  - Datenstruktur: Queue/Linked-List
- Shortest Job First
  - Braucht Zeitschätzung / Prioritäten
- Priorities
  - Datenstruktur: Zuordnung / Sorting
- Multilevel Queue
  - Datenstruktur: Prioritäten + Queue pro Prioritätswert