Universität Bielefeld – Technische Fakultät
AG Rechnernetze und verteilte Systeme

Vorl. 6: Single- und Multitasking

Wintersemester 2001/2002

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

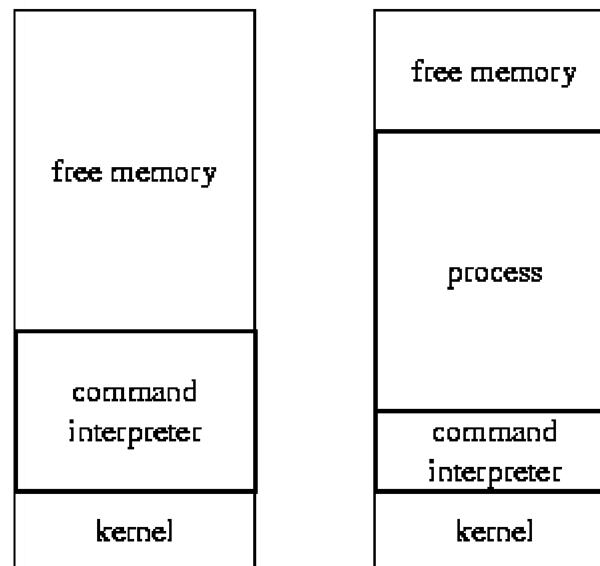
Single Tasking

- 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 Program muss
  "JMP <ComInt-Adresse>" am Ende ausführen

Single Tasking

- Einfach zu programmieren
- braucht keine Clock-Interrupts
- kann Busy-Wait auf I/O (warten muss er sowieso)
- Wenn ein Program stolpert, hält die Maschine
- Allgemeinstes Beispiel: MS-DOS

Single Tasking



Single-Tasking: MS-DOS

- Basis für Windows-3.x, Windows-95, 98 usw
- Die "persönliche" Versionen
- Aber nicht Windows-NT
- MS-DOS ist 16-Bit (im IBM-XT mit 8-Bit Datenübertragung wegen des Chipsatzes)

Multitasking

- Programme laufen "gleichzeitig"
- Ein Scheduler-Programm (BS) verteilt die Prozessor-Zeit auf die laufenden Programmen
- Ein Programm könnte also zweimal an der gleichen Zeit ausgeführt werden
- Wir sprechen nicht mehr von Programmen aber lieber von "Prozessen"

Zeit-Ablauf

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

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

endloop
```

Prozessen

- Ein Prozess ist ein "Programm im Lauf"
- Besteht aus "aktuellen Zustand" (Current State)
- Current State beinhält Werte von PC und alle Register
- Weiter, die Werte von den Program-Variablen
- Weiter, ist das Programm Ready, Waiting, Running, Terminated?

Process Table

- Die Prozess-Zustände werden in einer Tabelle gespeichert: Die Process Table
- Die Process Table beinhält die 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 Prozessen müssen zugeordnet werden
- Wenn ein I/O Interrupt signaliziert, dass die I/O fertig ist, wird der Handler den entsprechenden Prozess von Waiting zu Ready umstellen
- Wenn ein Prozess nur wegen Zeitablauf ausgeswapped 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:

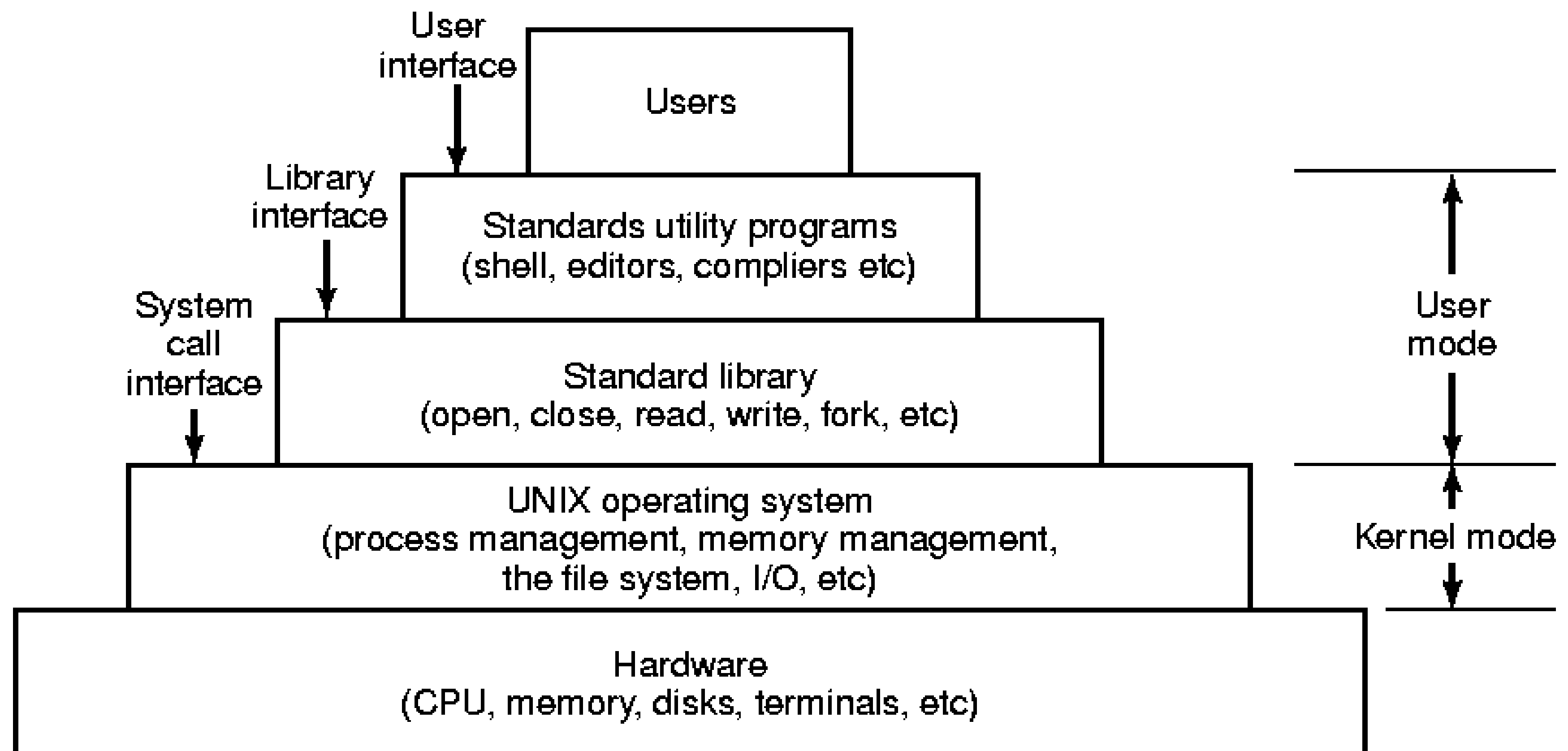
    Store(CurrentState, ProcTab.WaitingQ)

    Load(Head(Processtable.ReadyQueue)

Queues

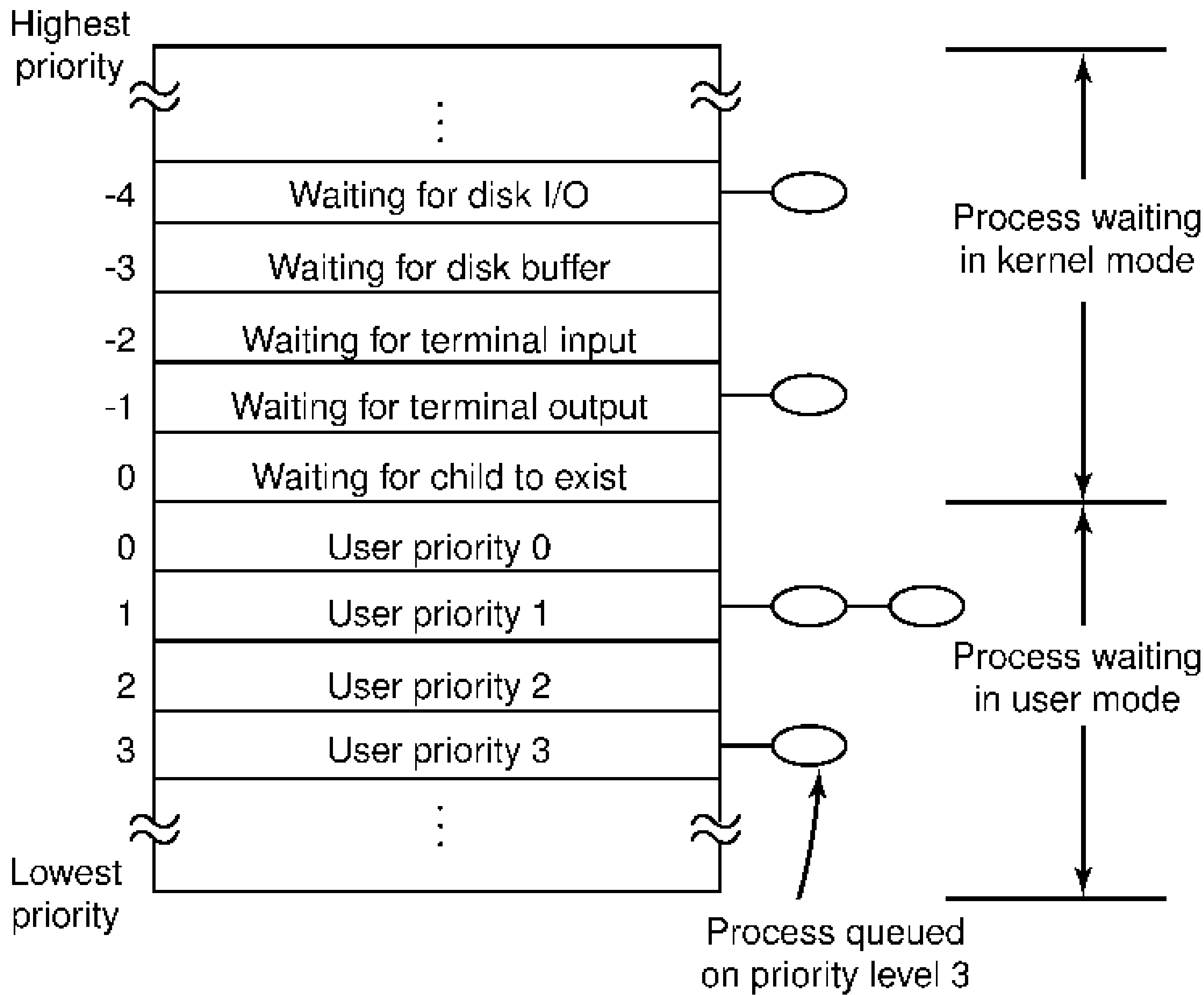
- Verwaltung von Ready-Queue ist nicht unbedingt einfach
- Ready-Prozessen könnte klassifiziert werden
  - Im nächsten Bild, eine 5-fache Klassifizierung

Prozess Klassifizierung



Prozess Klassifizierung

- Im nächsten Bild, wie die Process Table mit Prioritäten, sowie auch Begründungen für Waiting-Status (auch eine Art Klassifizierung) aussieht


Highest priority
-4
Waiting for disk I/O
-3
Waiting for disk buffer
-2
Waiting for terminal input
-1
Waiting for terminal output
0
Waiting for child to exist
0
User priority 0
1
User priority 1
2
User priority 2
3
User priority 3
Lowest priority
Process waiting in kernel mode
Process waiting in user mode
Process queued on priority level 3

Prozess Klassifizierung

- Im nächsten Bild, eine feinere Klassifizierung der Betriebssystem-Software

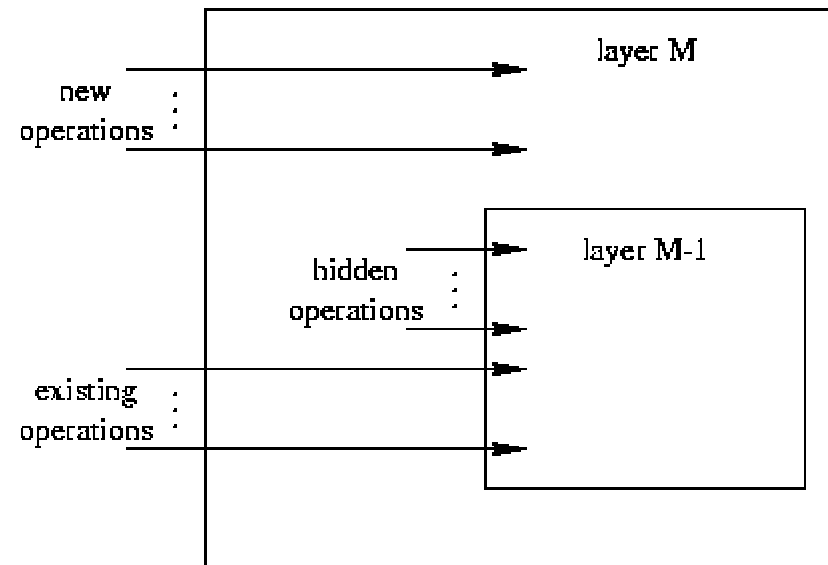
Prozess Klassifizierung

(the users)		
shells and commands compilers and interpreters system libaries		
system-call interface to the kernel		
signals terminal handling character I/O system terminal drivers	file system swapping block I/O system disk and tape drivers	CPU scheduling page replacement demand paging virtual memory
kernel interface to the kernel		
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

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



Ready Prozess Auswahl

- 3 Ready-Prozessen, 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 der Scheduling

- Wir haben **Starvation** gesehen
- **Ausserdem gibt es Deadlock** -
  - **Zwei oder mehr Prozessen warten gegenzeitig aufeinander (d.h. der eine wartet, bis der andere was tut, und der andere wartet, bis der erste was tut)**
- **Die zwie grösste Probleme der Scheduling**
- **Scheduling ist nicht trivial**

Beispiel: Scheduling Policy I

- Jeder *Ready* Prozess wird gestartet und läuft bis zu 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

- Fragen wir, wann welche Policy vernünftig wird
- Policy I
  - Einfache Systeme mit beschränkten Ressourcen
  - Ein-Benutzer Systeme
  - Echtzeit Systeme, in denen jeder Prozess und dessen genaue Zeitlauf ist bekannt, vertraut, wichtig und Hard-Deadlined
  - Naturwissenschaftliche Supercomputer

Scheduling Policy Auswahl

- Wann wird Policy II vernünftig?
  - 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