Kernel Architectures

Standard way: *monolithic kernel*:

Only two levels: user mode and kernel mode
All kernel code executed in kernel mode with full privileges
Example: Linux

Microkernel

Idea: Restrict amount of code running in kernel mode to minimum
⇒ Implement remainder of OS as services

At bottom: have microkernel with functions like

- Memory Management
- Scheduling
- Low-level device drivers

Higher-level parts like filesystems implemented in user space

Communication between parts of OS

Message passing used
Often combined with capabilities for good permission handling
⇒ Efficient message passing vital for performance
Message passing lends itself to asynchronous communication
⇒ bad for implementing Unix system calls
Suitable for embedded systems, in particular special real-time OS

Embedded Systems

Chips with power of whole computer systems now in many applications:

- Mobile Phones
- PDA's
- Smart Cards
- On-board controllers of HW

Characterisation of those systems:

- Fewer resources available: memory, storage space
- Often real-time applications necessary (on-board controllers)

Limited Resources

Not so much of a problem in general: OS's designed for this case
Only issue: potentially missing MMU
⇒ virtual memory and protection of processes against each other not implementable
Also paging not available

Real-time Operating Systems

Have two different kinds of real-time
1.) Hard real-time: completion required within a guaranteed amount of time
cannot be met by normal time-sharing systems; needs dedicated HW and adaptations to SW
2.) soft real-time: critical processes receive priority. Requires

- pre-emptive priority scheduling (plus sufficient resources to avoid starvation)
- short dispatch latency (time between arrival of process and start of execution)
  Problem: context switch normally only after syscall-completion or when I/O takes place
  way out: make kernel pre-emptible (eg Solaris 2, newer versions of Linux)