Operating Systems '2016

Jürgen Schönwälder

JACOBS UNIVERSITY

November 28, 2016

http://cnds.eecs.jacobs-university.de/courses/os-2016/

Part: Preface

Computer Networks & Distributed Systems

General Computer Science	1st Semester
Programming in C I (Jacobs Track — Skills)	1st Semester
Algorithms and Data Structures	2nd Semester
Programming in C II (Jacobs Track — Skills)	2nd Semester
Computer Architecture and Programming Languages	3rd Semester
Operating Systems	3rd Semester
Computer Networks	4th Semester

Course Content

- Introduction and Terminology
- Processes and Threads (Synchronization, Deadlocks)
- Memory Management (Segmentation, Paging)
- Virtual Memory
- Inter-Process Communication (Signals, Pipes, Sockets)
- Block and Character Devices
- File Systems
- Virtualization and Virtual Machines
- Embedded Operating Systems

Course Objectives

- Understand how an operating systems manages to turn a collection of independent hardware components into a useful abstraction
- Understand concurrency issues and be able to solve synchronization problems
- Knowledge about basic resource management algorithms
- Understand how memory management systems work and how they may impact the performance of systems
- Basic knowledge of inter-process communication mechanisms (signals, pipes, sockets)
- Understand some tradeoffs in the design of filesystems
- Learn about virtualization and embedded operating systems

Grading Scheme

- Final examination (40%)
  - Covers the whole lecture
  - Closed book (and closed computers / networks)
- Quizzes (30%)
  - Control your continued learning success
  - 6 quizzes with 10 pts each
  - 50 pts and above equals 30% of the overall grade
- Assignments (30%)
  - Individual submission of homework solutions
  - Homeworks sometimes include programming assignments
  - 6 assignments with 10 pts each
  - 50 pts and above equals 30% of the overall grade

Reading Material

A. Silberschatz, P. Galvin, and G. Gagne.
Applied Operating System Concepts.
John Wiley and Sons, 1 edition, 2000.

A. S. Tanenbaum and H. Bos.
Modern Operating Systems.
Pearson, 4 edition, 2015.

W. Stallings.
Operating Systems: Internals and Design Principles.
Pearson, 8 edition, 2014.

R. Love.
Linux Kernel Development.
Addison-Wesley, 3 edition, 2010.

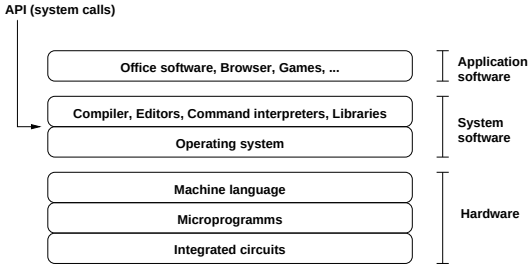
Part: Introduction

5. Definition and Requirements/Services
6. Types of Operating Systems
7. Operating System Architectures
8. Evolution of Operating Systems

What is an Operating System?

- An operating system is similar to a government... Like a government, the operating system performs no useful function by itself. (A. Silberschatz, P. Galvin)
- The most fundamental of all systems programs is the operating system, which controls all the computer's resources and provides the basis upon which the application programs can be written. (A.S. Tanenbaum)
- Examples:
  - Solaris, HP-UX, AIX, Linux, BSD, MAC OS X
  - Windows (Microsoft), MAC OS (Apple), OS/2 (IBM)
  - MVS (IBM), OS/390 (IBM), BS 2000 (Siemens)
  - VxWorks, Embedded Linux, Embedded BSD, Symbian
  - TinyOS, Contiki, RIOT

Hardware vs. System vs. Application



- From the operating system perspective, the hardware is mainly characterized by the machine instruction set.
- The operating system is part of the system software which includes system libraries and tools.
- Applications are build on top of the system software.

General Requirements

- An operating system should manage resources in a way which avoids shortages or overload conditions
- An operating system should be efficient and introduce little overhead
- An operating system should be robust against malfunctioning application programs
- Data and programs should be protected against unauthorized access and hardware failures

$\Longrightarrow$ Some of the requirements are contradictory

Services for Application Programs

- Loading of programs
- Execution of programs (management of processes)
- High-level input/output operations
- Logical file systems (`open()`, `write()`, ...)
- Control of peripheral devices
- Interprocess communication primitives
- Network interfaces
- Checkpoint and restart primitives
- ...

Services for System Operation

- User identification and authentication
- Access control mechanisms
- Encryption support
- Administrative functions (e.g., abort of processes)
- Test functions (e.g., detection of bad sectors on disks)
- System configuration and monitoring functions
- Event logging functions
- Collection of usage statistics (accounting)
- System generation and backup functions
- ...

Types of Operating Systems

- Batch processing operating systems
- General purpose operating systems
- Parallel operating systems
- Distributed operating systems
- Real-time operating systems
- Embedded operating systems

Batch Processing Operating Systems

- Batch jobs are processed sequentially from a job queue
- Job inputs and outputs are saved in files or printed
- No interaction with the user during the execution of a batch program

General Purpose Operating Systems

- Multiple programs execute simultaneously (multi-programming, multi-tasking)
- Multiple users can use the system simultaneously (multi-user)
- Processor time is shared between the running processes (time-sharing)
- Input/output devices operate concurrently with the processor(s)
- Network support but no or very limited transparency
- Examples:
  - Linux, BSD, Solaris, ...
  - Windows, MacOS, ...

Parallel Operating Systems

- Support for a large number of tightly integrated processors
- Symmetrical
  - Each processor has a full copy of the operating system
- Asymmetrical
  - Only one processor carries the full operating system
  - Other processors are operated by a small operating system stub to transfer code and tasks

Distributed Operating Systems

- Support for a medium number of loosely coupled processors
- Processors execute a small operating system kernel providing essential communication services
- Other operating system services are distributed over available processors
- Services can be replicated in order to improve scalability and availability
- Distribution transparent to users (single system image)
- Examples:
  - Amoeba (Vrije Universiteit Amsterdam) [**?**]

Real-time Operating Systems

- Predictability
- Logical correctness of the offered services
- Timeliness of the offered services
- Services are to be delivered not too early, not too late
- Operating system switches processes based on time constraints
- Required for robots, medical devices, communication control in vehicles, ...
- Examples:
  - QNX
  - VxWorks
  - RTLinux
  - Windows CE

Embedded Operating Systems

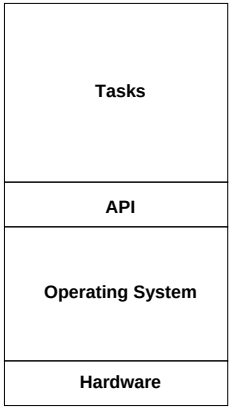
- Usually real-time systems, often hard real-time systems
- Very small memory footprint (even today!)
- None or limited user interaction
- 90-95 % of all processors are running embedded operating systems
- Special variants of Linux and BSD systems are being developed to support embedded systems and gaining momentum
- Examples:
  - Embedded Linux, Embedded BSD
  - Symbian OS, Windows Mobile, iPhone OS, BlackBerry OS, Palm OS
  - Cisco IOS, JunOS, IronWare, Inferno
  - Contiki, TinyOS, RIOT

Operating System Architectures

- Monolithic operating systems
- Monolithic modular operating systems
- Monolithic layered operating systems
- Virtual machines
- Client/Server operating systems
- Distributed operating systems

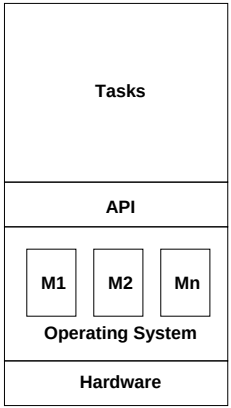
Monolithic Operating Systems

- A collection of functions without a structure (the big mess)
- Typically non-portable, hard to maintain, lack of reliability
- All services are in the kernel with the same privilege level
- Monolithic systems can be highly efficient



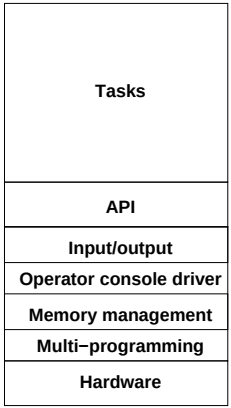
Monolithic & Modular Operating Systems

- Modules can be platform independent
- Easier to maintain and to develop
- Increased reliability / robustness
- All services are in the kernel with the same privilege level
- May reach high efficiency
- Example: Linux



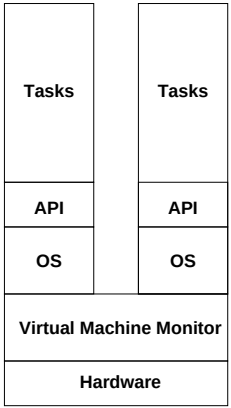
Monolithic & Layered Operating Systems

- Easily portable, significantly easier to maintain
- Likely reduced efficiency through many layered interfaces
- Rigorous implementation of the stacked virtual machine perspective
- Services offered by the various layers are important for the performance
- Example: THE (Dijkstra, 1968)



Virtual Machines

- Virtualization of the hardware
- Multiple operating systems can execute concurrently
- Separation of multi-programming from other operating system services
- Examples: IBM VM/370 ('79), VMware (1990s), XEN (2003)
- Related: User Mode Linux, Linux Kernel Based Virtual Machine, Linxu VServer



Evolution of Operating Systems

- 1st Generation (1945-1955): Vacuum Tubes
  - Manual operation, no operating system
  - Programs are entered via plugboards
- 2nd Generation (1955-1965): Transistors
  - Batch systems automatically process jobs
  - The list of jobs is stored on magnetic tapes
- 3rd Generation (1965-1980): Integrated Circuits
  - Spooling (Simultaneous Peripheral Operation On Line)
  - Multiprogramming and Time-sharing
  - OS/360 (IBM), Multics (MIT, Bell Labs, General Electric)
- 4th Generation (1980-2000): VLSI
  - Personal computer (CP/M, MS-DOS, Windows, Mac OS, Unix)
  - Network operating systems (Unix)
  - Distributed operating systems (Amoeba, Mach, V)

References

Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren.
Amoeba, A distributed operating system for the 1990s.
Computer, 23(5):44–53, May 1990.

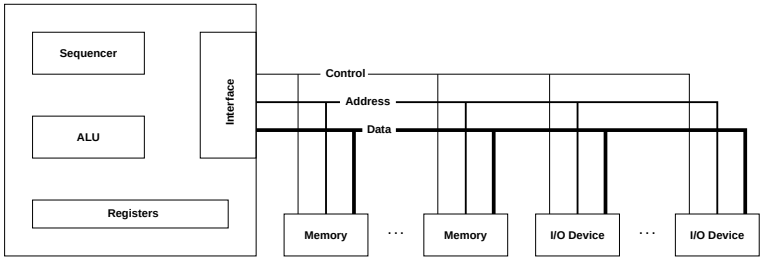
M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman.
The Design and Implementation of the 4.4 BSD Operating System.
Addison Wesley, 1996.

R. Love.
Linux Kernel Development.
Sams Publishing, 2003.

P. H. Salus.
A Quarter Century of UNIX.
Addison Wesley, 1994.

Part: Hardware

Common Computer Architecture



- Common computer architecture uses busses to connect memory and I/O systems to the central processing unit
- Requires arbitration, synchronization, interrupts, priorities

CPU Registers

- Typical set of registers:
  - Processor status register
  - Instruction register (current instruction)
  - Program counter (current or next instruction)
  - Stack pointer (top of stack)
  - Special privileged registers
  - Dedicated registers
  - Universal registers
- Privileged registers are only accessible when the processor is in privileged mode
- Switch from non-privileged to privileged mode via traps or interrupts

CPU Instruction Sets

- Non-privileged instruction set:
  - General purpose set of processor instructions
- Privileged instruction set:
  - Provide access to special resources such as privileged registers or memory management units
  - Subsumes the non-privileged instruction set
- Some processors support multiple privilege levels
- Changes to higher privilege levels via traps / interrupts only

I/O Devices

- I/O devices are essential for every computer
- Typical classes of I/O devices:
  - clocks, timers
  - user-interface devices
  - document I/O devices (scanner, printer, ...)
  - audio and video equipment
  - network interfaces
  - mass storage devices
  - sensors and actuators in control applications
- Device drivers are often the biggest component of general purpose operating system kernels

Basic I/O Programming

- *Status driven*: the processor polls an I/O device for information
  - Simple but inefficient use of processor cycles
- *Interrupt driven*: the I/O device issues an interrupt when data is available or an I/O operation has been completed
  - *Program controlled*: Interrupts are handled by the processor directly
  - *Program initiated*: Interrupts are handled by a DMA-controller and no processing is performed by the processor (but the DMA transfer might steal some memory access cycles, potentially slowing down the processor)
  - *Channel program controlled*: Interrupts are handled by a dedicated channel device, which is usually itself a micro-processor

Interrupts

- Interrupts can be triggered by hardware and by software
- Interrupt control:
  - grouping of interrupts
  - encoding of interrupts
  - prioritizing interrupts
  - enabling / disabling of interrupt sources
- Interrupt identification:
  - interrupt vectors, interrupt states
- Context switching:
  - mechanisms for CPU state saving and restoring

Interrupt Service Routines

- Minimal hardware support (supplied by the CPU)
  - Save essential CPU registers
  - Jump to the vectorized interrupt service routine
  - Restore essential CPU registers on return
- Minimal wrapper (supplied by the operating system)
  - Save remaining CPU registers
  - Save stack-frame
  - Execute interrupt service code
  - Restore stack-frame
  - Restore CPU registers

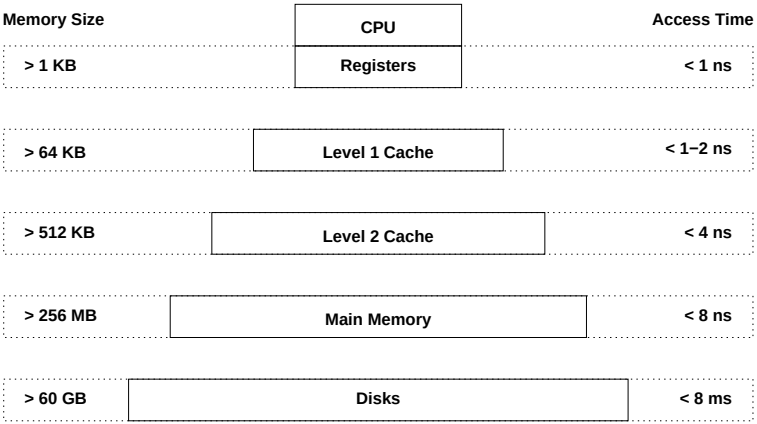
Interrupt Handling Sketch 1/2

```
void (*interrupt_handler)(void);

interrupt_handler interrupt_vector[] =
{
   handler_a,
   handler_b,
   ...
}

/* the following logic executed by the hardware when an interrupt *
 * has arrived and the execution of an instruction is complete    */

// on interrupt #x:
//   save_essential_registers();      // includes instruction pointer
//   handler = interrupt_vector[#x];
//   if (handler) handler();
//   restore_essential_registers();   // includes instruction pointer
```

Interrupt Handling Sketch 2/2

```
void handler_a(void)
{
   save_cpu_registers();
   save_stack_frame();
   interrupt_a_handling_logic();
   restore_stack_frame();
   restore_cpu_registers();
}

void handler_b(void)
{
   save_cpu_registers();
   save_stack_frame();
   interrupt_b_handling_logic();
   restore_stack_frame();
   restore_cpu_registers();
}
```

Memory Sizes and Access Times



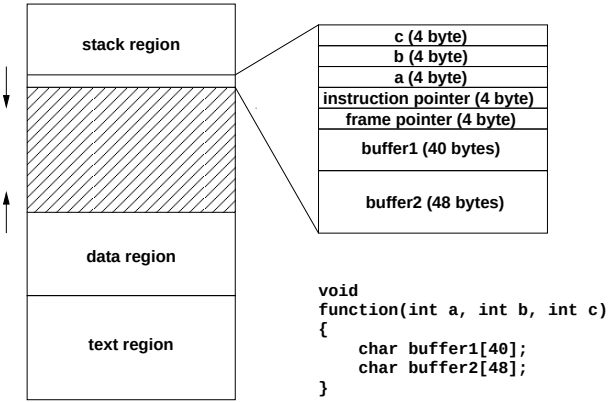
- There is a trade-off between memory speed and memory size.

Memory Segments

Segment	Description
text	machine instructions of the program
data	static variables and constants, may be further devided into initialized and uninitialized data
heap	dynamically allocated data structures
stack	automatically allocated local variables, management of function calls (parameters, results, return addresses)

- Memory used by a program is usually partitioned into different segments that serve different purposes and may have different access rights

Stack Frames



- Every function call leaves an entry (stack frame) on the stack
- Stack frame layout is processor specific (here Intel x86)

Example

```
static int foo(int a)
{
    static int b = 5;
    int c;

    c = a * b;
    b += b;
    return c;
}

int main(int argc, char *argv[])
{
    return foo(foo(1));
}
```

- What is returned by `main()`?
- Which segments store the variables?

Stack Smashing Attacks

```
#include <string.h>

void foo(char *bar)
{
    char c[12];
    strcpy(c, bar);  // no bounds checking
}

int main(int argc, char *argv[])
{
    if (argv[1]) foo(argv[1]);
    return 0;
}
```

- Overwriting of a function return address on the stack
- Returning into a 'landing area' (typically sequences of NOPs)
- Landing area is followed by shell code (code to start a shell)

Caching

- Caching is a general technique to speed up memory access by introducing smaller and faster memories which keep a copy of frequently / soon needed data
- *Cache hit*: A memory access which can be served from the cache memory
- *Cache miss*: A memory access which cannot be served from the cache and requires access to slower memory
- *Cache write through*: A memory update which updates the cache entry as well as the slower memory cell
- *Delayed write*: A memory update which updates the cache entry while the slower memory cell is updated at a later point in time

Locality

- Cache performance is relying on:
  - *Spatial locality*:
    Nearby memory cells are likely to be accessed soon
  - *Temporal locality*:
    Recently addressed memory cells are likely to be accessed again soon
- Iterative languages generate linear sequences of instructions (spatial locality)
- Functional / declarative languages extensively use recursion (temporal locality)
- CPU time is in general often spend in small loops/iterations (spatial and temporal locality)
- Data structures are organized in compact formats (spatial locality)

Part: Processes

0;95;0c

Entering the Operating System Kernel

- System calls (supervisor calls, software traps)
  - Synchronous to the running process
  - Parameter transfer via registers, the call stack or a parameter block
- Hardware traps
  - Synchronous to a running process (devision by zero)
  - Forwarded to a process by the operating system
- Hardware interrupts
  - Asynchronous to the running processes
  - Call of an interrupt handler via an interrupt vector
- Software interrupts
  - Asynchronous to the running processes

User Mode

- The processor executes machine instructions of (user) processes
- The instruction set of the processor and the set of accessible registers is restricted to the so called *unprivileged instruction set*
- Memory addresses used by a user process are typically mapped to physical memory addresses by the memory management unit.
- Direct access to the hardware components (e.g., memory) is protected by using hardware protection where possible
- Direct access to the state of other concurrently running processes is normally restricted

System Mode

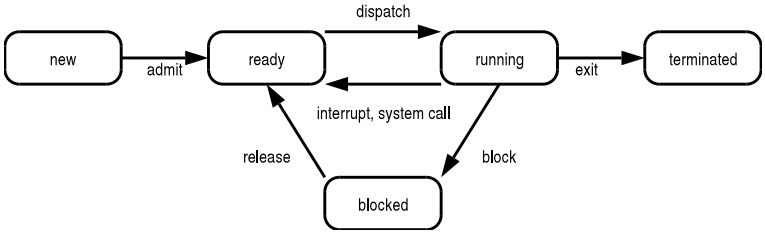
- The processor executes machine instructions of the operating system kernel
- All instructions and registers of the processor can be used, also called the *privileged instruction set*
- Access to physical memory addresses and the memory address mapping tables
- Direct access to the hardware components of the system
- Manipulation of the state of running processes possible

$\Longrightarrow$ The operating system generally runs in system mode while user processes execute in user mode.

Process Characterization

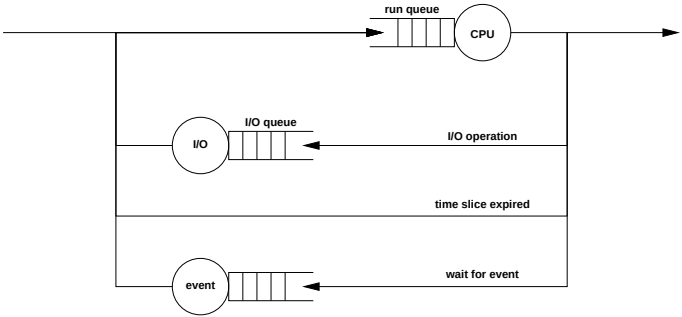
- A process is an instance of a program under execution
- A process uses/owns resources (e.g., CPU, memory, files) and is characterized by the following:
  1. A sequence of machine instructions which determines the behavior of the running program (control flow)
  2. The current state of the process given by the content of the processor's registers, the contents of the stack, and the contents of the heap (internal state)
  3. The state of other resources (e.g., open files or network connections, timer, devices) used by the running program (external state)
- Processes are sometimes also called tasks.

Processes: State Machine View



- *new*: just created, not yet admitted
- *ready*: ready to run, waiting for CPU
- *running*: executing, holds a CPU
- *blocked*: not ready to run, waiting for a resource
- *terminated*: just finished, not yet removed

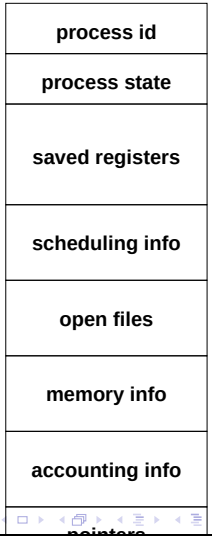
Processes: Queueing Model View



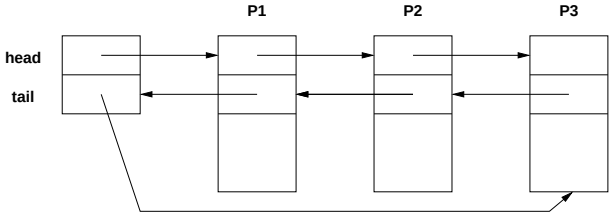
- Processes are enqueued if resources are not readily available or if processes wait for events
- Dequeuing strategies have strong performance impact
- Queueing models can be used for performance analysis

Process Control Block

- Processes are internally represented by a process control block (PCB)
  - Process identification
  - Process state
  - Saved registers during context switches
  - Scheduling information (priority)
  - Assigned memory regions
  - Open files or network connections
  - Accounting information
  - Pointers to other PCBs
- PCBs are often enqueued at a certain state of condition

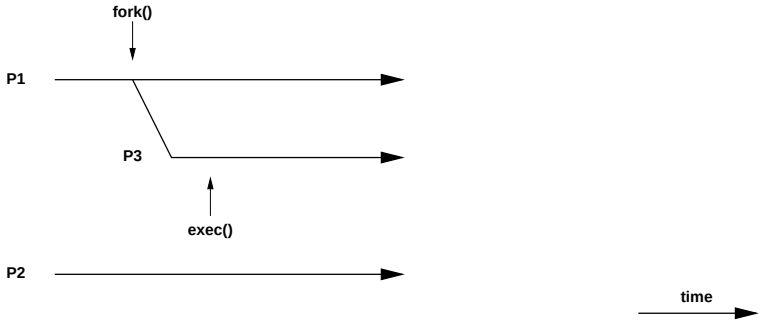


Process Lists



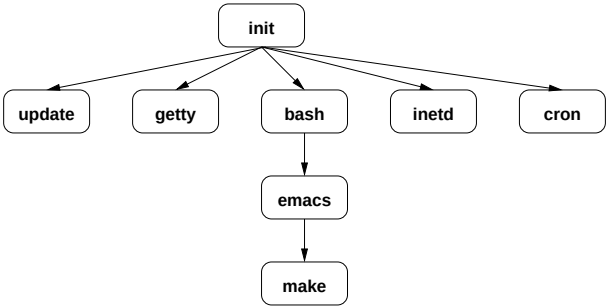
- PCBs are often organized in doubly-linked lists or tables
- PCBs can be queued by pointer operations
- Run queue length of the CPU is a good load indicator
- The system load often defined as the exponentially smoothed average of the run queue length over 1, 5 and 15 minutes

Process Creation



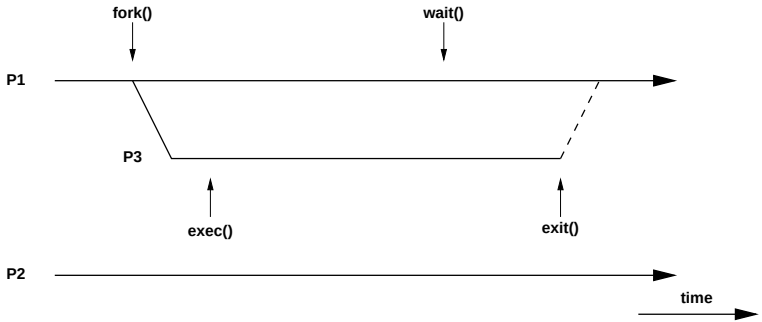
- The `fork()` system call creates a new child process
  - which is an exact copy of the parent process,
  - except that the result of the system call differs
- The `exec()` system call replaces the current process image with a new process image.

Process Trees



- First process is created when the system is initialized
- All other processes are created using `fork()`, which leads to a tree of processes
- PCBs often contain pointers to parent PCBs

Process Termination



- Processes can terminate themself by calling `exit()`
- The `wait()` system call allows processes to wait for the termination of a child process
- Terminating processes return a numeric result code

POSIX API (fork, exec)

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);

pid_t fork(void);
int execve(const char  *filename, char *const argv [],
           char *const envp[]);

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const  char  *path,  const  char  *arg, ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

POSIX API (exit, wait)

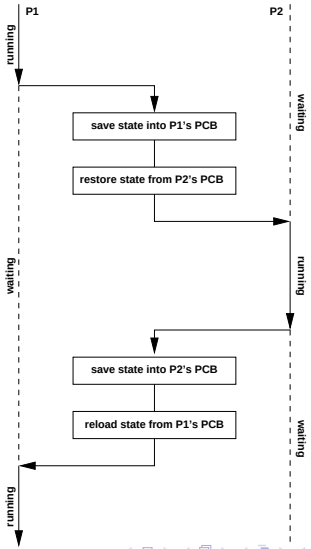
```
#include <stdlib.h>

void exit(int status);
int atexit(void (*function)(void));

#include <unistd.h>

void _exit(int status);

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

Sketch of a Command Interpreter

```
while (1) {
    show_prompt();                  /* display prompt */
    read_command();                 /* read and parse command */
    pid = fork();                   /* create new process */
    if (pid < 0) {                  /* continue if fork() failed */
        perror("fork");
        continue;
    }
    if (pid != 0) {                 /* parent process */
        waitpid(pid, &status, 0);   /* wait for child to terminate */
    } else {                        /* child process */
        execvp(args[0], args, 0);   /* execute command */
        perror("execvp");           /* only reach on exec failure */
        _exit(1);                   /* exit without any cleanups */
    }
}
```

Context Switch

- Save the state of the running process/thread
- Reload the state of the next running process/thread
- Context switch overhead is an important operating system performance metric
- Switching processes can be expensive if memory must be reloaded
- Preferable to continue a process or thread on the same CPU



Threads

- Threads are individual control flows, typically within a process (or within a kernel)
- Every thread has its own private stack (so that function calls can be managed for each thread separately)
- Multiple threads share the same address space and other resources
  - Fast communication between threads
  - Fast context switching between threads
  - Often used for very scalable server programs
  - Multiple CPUs can be used by a single process
  - Threads require synchronization (see later)
- Some operating systems provide thread support in the kernel while others implement threads in user space

POSIX API (pthreads)

```
#include <pthread.h>

typedef ... pthread_t;
typedef ... pthread_attr_t;

int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void * (*start) (void *),
                   void *arg);
void pthread_exit(void *retval);
int pthread_cancel(pthread_t thread);
int pthread_join(pthread_t thread, void **retvalp);

int pthread_cleanup_push(void (*func)(void *), void *arg)
int pthread_cleanup_pop(int execute)
```

Processes and Threads in Linux (2.6.x)

- Linux internally treats processes and threads as so called tasks
- Linux distinguishes three different types of tasks:
  1. idle tasks (also called idle threads)
  2. kernel tasks (also called kernel threads)
  3. user tasks
- Tasks are in one of the states *running*, *interruptible*, *uninterruptible*, *stopped*, *zombie*, or *dead*
- A special `clone()` system call is used to create processes and threads

Processes and Threads in Linux (2.6.x)

- Linux tasks (processes) are represented by a `struct task_struct` defined in `<linux/sched.h>`
- Tasks are organized in a circular, doubly-linked list with an additional hashtable, hashed by process id (pid)
- Non-modifying access to the task list requires the usage of the `tasklist_lock` for `READ`
- Modifying access to the task list requires the usage the `tasklist_lock` for `WRITE`
- System calls are identified by a number
- The `sys_call_table` contains pointers to functions implementing the system calls

Race Conditions

- A *race condition* exists if the result produced by concurrent processes (or threads), which access and manipulate shared resources (variables), depends unexpectedly on the order of the execution of the processes (or threads)

$\Longrightarrow$ Protection against race conditions is a very important issue within operating system kernels, but equally well in many application programs

$\Longrightarrow$ Protection against race conditions is difficult to test (actual behaviour usually depends on many factors that are hard to control)

$\Longrightarrow$ High-level programming constructs move the generation of correct low-level protection into the compiler

Bounded Buffer Problem

- Two processes share a common fixed-size buffer
- The producer process puts data into the buffer
- The consumer process reads data out of the buffer
- The producer must wait if the buffer is full
- The consumer must wait if the buffer is empty

```
void producer()
{
    produce(&item);
    while (count == N) sleep(1);
    buffer[in] = item;
    in = (in + 1) % N;
    count = count + 1;
}
```

```
void consumer() {
{
     while (count == 0) sleep(1);
     item = buffer[out];
     out = (out + 1) % N;
     count = count - 1;
     consume(item);
 }
```

$\Longrightarrow$ This solution has a race condition and is not correct!

Bounded Buffer Problem

- Pseudo machine code for `count = count + 1` and for `count = count - 1`:

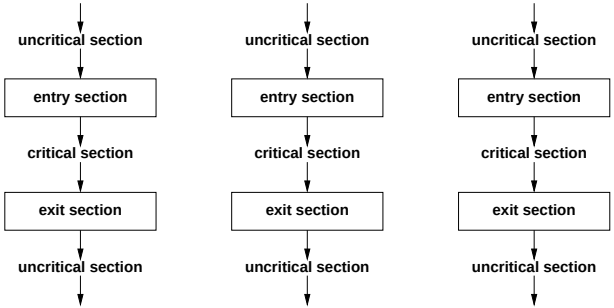
```
P1: load   reg_a,count          C1: load   reg_b,count
P2: incr   reg_a                C2: decr   reg_b
P3: store  reg_a,count          C3: store  reg_b,count
```

- Lets assume `count` has the value 5. What happens to `count` in the following execution sequences?
  a) `P1, P2, P3, C1, C2, C3` leads to the value 5
  b) `P1, P2, C1, C2, P3, C3` leads to the value 4
  c) `P1, P2, C1, C2, C3, P3` leads to the value 6

$\Longrightarrow$ Every situation, in which multiple processes (threads) manipulate shared resources, can lead to race conditions

$\Longrightarrow$ Synchronization mechanisms are always needed to coordinate access to shared resources

Critical Sections



- A *critical section* is a segment of code that can only be executed by one process at a time
- The execution of critical sections by multiple processes is *mutually exclusive* in time
- Entry and exit sections must protect critical sections

Critical-Section Problem

- The *critical-section problem* is to design a protocol that the processes can use to cooperate
- A solution must satisfy the following requirements:
  1. *Mutual Exclusion*: No two processes may be simultaneously inside the same critical section.
  2. *Progress*: No process outside its critical sections may block other processes.
  3. *Bounded-Waiting*: No process should have to wait forever to enter its critical section.
- General solutions are not allowed to make assumptions about execution speeds or the number of CPUs present in a system.

Disabling Interrupts

- The simplest solution is to disable all interrupts during the critical section so that nothing can interrupt the critical section

```
disable_interrupts();
critical_section();
enable_interrupts();
```

- Can usually not be used in user-space
- Problematic on systems with multiple processors
- Fails if interrupts are needed in the critical section
- Very efficient and sometimes used in some special cases

Strict Alternation

- Lets assume just two processes which share a variable called `turn` which holds the values 0 and 1

```
/* process 0 */                    /* process 1 */
uncritical_section();              uncritical_section();
while (turn != 0) sleep(1);        while (turn != 1) sleep(1);
criticial_section();               critical_section()
turn = 1;                          turn = 0;
uncritical_section();              uncritical_section();
```

- Ensures mutual exclusion
- Can be extended to handle alternation between N processes
- Does not satisfy the progress requirement since the solution enforces strict alternation

Peterson's Algorithm

- Lets assume two processes *i* and *j* sharing a variable `turn` (which holds a process identifier) and a boolean array `interested`, indexed by process identifiers

```
uncritical_section();
interested[i] = true;
turn = j;
while (interested[j] && turn == j) sleep(1);
criticial_section();
interested[i] = false;
uncritical_section();
```

- Modifications of `turn` (and `interested`) are protected by a loop to handle concurrency issues
- Algorithm satisfies mutual exclusion, progress and bounded-waiting requirements and can be extended to handle N processes

Spin-Locks

- So called *spin-locks* are locks which cause the processor to spin while waiting for the lock
- Spin-locks are often used to synchronize multi-processor systems with shared memory
- Spin-locks require atomic test-and-set-lock machine instructions on shared memory cells
- Reentrant locks do not harm if you already hold a lock

```
enter:  tsl     register, flag  ; copy shared variable flag to register and set flag to 1
        cmp     register, #0    ; was flag 0?
        jnz     enter           ; if not 0, a lock was set, so try again
        ret                     ; critical region entered

leave:  move    flag, #0        ; clear lock by storing 0 in flag
        ret                     ; critical region left
```

Critique

- Busy waiting potentially wastes processor cycles
- Busy waiting can lead to *priority inversion*:
  - Consider processes with high and low priority
  - Processes with high priority are preferred over processes with lower priority by the scheduler
  - Once a low priority process enters a critical section, processes with high priority will be slowed down more or less to the low priority
  - Depending on the scheduler, complete starvation is possible

$\Longrightarrow$ Find alternatives which do not require busy waiting

Semaphores

- A *semaphore* is a protected integer variable which can only be manipulated by the atomic operations `up()` and `down()`
- High-level definition of the *behavior* of semaphores:

```
down(s)
{
    s = s - 1;
    if (s < 0) queue_this_process_and_block();
}

up(s)
{
    s = s + 1;
    if (s <= 0) dequeue_and_wakeup_process();
}
```

- Dijkstra called the operations `P()` and `V()`, other popular names are `wait()` and `signal()`

Critical Sections with Semaphores

- Critical sections are easy to implement with semaphores:

```
semaphore mutex = 1;
```

```
uncritical_section();
down(&mutex);
critical_section();
up(&mutex);
uncritical_section();
```

```
uncritical_section();
down(&mutex);
critical_section();
up(&mutex);
uncritical_section();
```

- Rule of thumb: Every access to a shared data object must be protected by a `mutex` semaphore for the shared data object as shown above
- However, some synchronization problems require more creative usage of semaphores for proper coordination

Bounded Buffer with Semaphores

```
const int N;
shared item_t buffer[N];
semaphore mutex = 1, empty = N, full = 0;
```

```
void producer()
{
    produce(&item);
    down(&empty);
    down(&mutex);
    buffer[in] = item;
    in = (in + 1) % N;
    up(&mutex);
    up(&full);
}
```

```
void consumer()
{
   down(&full);
   down(&mutex);
   item = buffer[out];
   out = (out + 1) % N;
   up(&mutex);
   up(&empty);
   consume(item);
}
```

- Semaphore `mutex` protects the critical section
- Semaphore `empty` counts empty buffer space
- Semaphore `full` counts used buffer space

Readers / Writers Problem

- A data object is to be shared among several concurrent processes
- Multiple processes (the readers) should be able to read the shared data object simultaneously
- Processes that modify the shared data object (the writers) may only do so if no other process (reader or writer) accesses the shared data object
- Several variations exist, mainly distinguishing whether either reader or writers gain preferred access

$\Longrightarrow$ Starvation can occur in many solutions and is not taken into account here

Readers / Writers with Semaphores

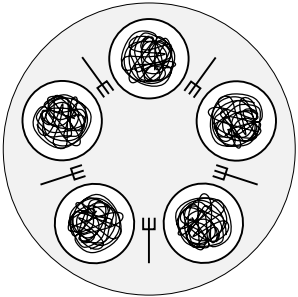
```
shared object data;
shared int readcount = 0;
semaphore mutex = 1, writer = 1;
```

```
void reader()
{
    down(&mutex);
    readcount = readcount + 1;
    if (readcount == 1) down(&writer);
    up(&mutex);
    read_shared_object(&data);
    down(&mutex);
    readcount = readcount - 1;
    if (readcount == 0) up(&writer);
    up(&mutex);
}
```

```
void writer()
{
    down(&writer);
    write_shared_object(&data);
    up(&writer);
}
```

$\Longrightarrow$ Many readers can cause starvation of writers

Dining Philosophers



- Philosophers sitting on a round table either think or eat
- Philosophers do not keep forks while thinking
- A philosopher needs two forks (left and right) to eat
- A philosopher may not pick up only one fork at a time

Dining Philosophers with Semaphores

```
const int N;                /* number of philosophers  */
shared int state[N];        /* thinking (default), hungry or eating */
semaphore mutex = 1;        /* mutex semaphore to protect state */
semaphore s[N] = 0;         /* semaphore for each philosopher */
```

```
void philosopher(int i)
{
    while (true) {
        think(i);
        take_forks(i);
        eat(i);
        put_forks(i);
    }
}
```

```
void test(int i)
{
    if (state[i] == hungry
        && state[(i-1)%N] != eating
        && state[(i+1)%N] != eating) {
        state[i] = eating;
        up(&s[i]);
    }
}
```

- The `test()` function tests whether philosopher `i` can eat and conditionally unblocks his semaphore

Dining Philosophers with Semaphores

```
void take_forks(int i)
{
    down(&mutex);
    state[i] = hungry;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

```
void put_forks(int i)
{
    down(&mutex);
    state[i] = thinking;
    test((i-1)%N);
    test((i+1)%N);
    up(&mutex);
}
```

- The function `take_forks()` introduces a hungry state and waits for the philosopher's semaphore
- The function `put_forks()` gives the neighbors a chance to eat
- Starvation of philosophers? Fairness?
- What about trying to pick forks after waiting randomly?

Implementation of Semaphores

- The semaphore operations `up()` and `down()` must be atomic
- On uniprocessor machines, semaphores can be implemented by either disabling interrupts during the `up()` and `down()` operations or by using a correct software solution (e.g., Peterson's algorithm)
- On multiprocessor machines, semaphores are usually implemented by using spin-locks, which themself use special machine instructions
- Semaphores are therefore often implemented on top of more primitive synchronization mechanisms

Binary Semaphores

- Binary semaphores are semaphores that only take the two values 0 and 1.
- Counting semaphores can be implemented by means of binary semaphores:

```
shared int c;
binary_semaphore mutex = 1, wait = 0, barrier = 1;

void down()
{
    down(&barrier);
    down(&mutex);
    c = c - 1;
    if (c < 0) {
        up(&mutex);
        down(&wait);
    } else {
        up(&mutex);
    }
    up(&barrier);
}
```

```
void up()
{
    down(&mutex);
    c = c + 1;
    if (c <= 0) {
        up(&wait);
    }
    up(&mutex);
}
```

Semaphore Pattern: Mutual Exclusion

A critical section may only be executed by a single thread.

```
semaphore_t s = 1;

thread()
{
  /* do something */
  down(&s);
  /* critical section */
  up(&s);
  /* do something */
}
```

Semaphore Pattern: Multiplex

A section may be executed concurrently with a certain fixed limit of N concurrent threads. (This is a generalization of the mutual exclusion pattern, which is essentially multiplex with N = 1.)

```
semaphore_t s = N;

thread()
{
  /* do something */
  down(&s);
  /* multiplex section */
  up(&s);
  /* do something */
}
```

Semaphore Pattern: Signaling

A thread waits until some other thread signals a certain condition.

```
semaphore_t s = 0;

waiting_thread()
{
  /* do something */
  down(&s);
  /* do something */
}
```

```
signaling_thread()
{
  /* do something */
  up(&s);
  /* do something */
}
```

Semaphore Pattern: Rendezvous

Two threads wait until they both have reached a certain state (the rendezvous point) and afterwards they proceed independently again. (This can be seen as using the signaling pattern twice.)

```
semaphore_t s1 = 0, s2 = 0;

thread_A()                    thread_B()
{                             {
  /* do something */            /* do something */
  up(&s2);                      up(&s1);
  down(&s1);                    down(&s2);
  /* do something */            /* do something */
}
```

Semaphore Pattern: Simple Barrier

Generalization of the rendezvous pattern to N threads. First a simple barrier solution using counting semaphores.

```
shared int count = 0;
semaphore_t mutex = 1, turnstile = 0;

thread()
{
  /* do something */
  down(&mutex);
  count++;
  if (count == N) {
    for (int j = 0; j < N; j++) {
      up(&turnstile);          /* let N threads pass through the turnstile */
    }
    count = 0;
  }
  up(&mutex);
  down(&turnstile);            /* block until opened by the Nth thread */
  /* do something */
}
```

Semaphore Pattern: Double Barrier

Next a solution allowing to do something while passing through the barrier, which is sometimes needed.

```
shared int count = 0;
semaphore_t mutex = 1, turnstile1 = 0, turnstile2 = 1;

{
  /* do something */

  down(&mutex);
  count++;
  if (count == N) {
    down(&turnstile2);        /* close turnstile2 (which was left open) */
    up(&turnstile1);          /* open turnstile1 for one thread */
  }
  up(&mutex);
  down(&turnstile1);          /* block until opened by the last thread */
  up(&turnstile1);            /* every thread lets another thread pass */

  /* do something controlled by a barrier */
```

Semaphore Pattern: Double Barrier (cont.)

```
  /* do something controlled by a barrier */

  down(&mutex);
  count--;
  if (count == 0) {
    down(&turnstile1);          /* close turnstile1 again */
    up(&turnstile2);            /* open turnstile2 for one thread */
  }
  up(&mutex);
  down(&turnstile2);            /* block until opened by the last thread */
  up(&turnstile2);              /* every thread lets another thread pass */
                                /* (turnstile2 is left open) */

  /* do something */
}
```

Critical Regions

- Simple programming errors (omissions, permutations) with semaphores usually lead to difficult to debug synchronization errors
- Idea: Let the compiler do the tedious work

```
shared struct buffer {
    item_t pool[N]; int count; int in; int out;
}
```

```
region buffer when (count < N)
{
    pool[in] = item;
    in = (in + 1) % N;
    count = count + 1;
}
```

```
region buffer when (count > 0)
{
    item = pool[out];
    out = (out + 1) % N;
    count = count - 1;
}
```

- Reduces the number of synchronization errors, does not eliminate synchronization errors

Monitors

- Idea: Encapsulate the shared data object and the synchronization access methods into a monitor
- Processes can call the procedures provided by the monitor
- Processes can not access monitor internal data directly
- A monitor ensures that only one process is active in the monitor at every given point in time
- Monitors are special programming language constructs
- Compilers generate proper synchronization code
- Monitors were developed well before object-oriented languages became popular

Condition Variables

- Condition variables are special monitor variables that can be used to solve more complex coordination and synchronization problems
- Condition variables support the two operations `wait()` and `signal()`:
  - The `wait()` operation blocks the calling process on the condition variable `c` until another process invokes `signal()` on `c`. Another process may enter the monitor
  - The `signal()` operation unblocks a process waiting on the condition variable `c`. The calling process must leave the monitor
- Condition variables are not counters. A `signal()` on `c` is ignored if no processes wait on `c`

Bounded Buffer with Monitors

```
monitor BoundedBuffer
{
      condition full, empty;
      int count = 0;
      item_t buffer[N];

  void enter(item_t item)
  {
      if (count == N) wait(&full);
      buffer[in] = item;
      in = (in + 1) % N;
      count = count + 1;
      if (count == 1) signal(&empty);
  }

item_t remove()
{
    if (count == 0) wait(&empty);
    item = buffer[out];
    out = (out + 1) % N;
    count = count - 1;
    if (count == N-1) signal(&full);
    return item;
}

}
```

Messages

- Exchange of messages can be used for synchronization
- Two primitive operations:

  ```
  send(destination, message)
  recv(source, message)
  ```

- Blocking message systems block processes in these primitives if the peer is not ready for a rendevous
- Storing message systems maintain messages in special mailboxes called message queues. Processes only block if the remote mailbox is full during a `send()` or the local mailbox is empty during a `recv()`
- Some programming languages (e.g., go) use message queues as the primary abstraction for thread synchronization (e.g., go routines and channels)

Messages

- Message systems support the synchronization of processes that do not have shared memory
- Message systems can be implemented in user space and without special compiler support
- Message systems usually require that
  - messages are not lost during transmission
  - messages are not duplicated during transmission
  - addresses are unique
  - processes do not send arbitrary messages to each other
- Message systems are often slower than shared memory mechanisms
- POSIX message queues provide synchronization between threads or processes

Bounded Buffer with Messages

- Messages are used as tokens which control the exchange of messages
- Consumers initially generate and send a number of tokens to the producers

```
void init() { for (i = 0; i < N; i++) { send(&producer, &m); } }
```

```
void producer()
{
    produce(&item);
    recv(&consumer, &m);
    pack(&m, item);
    send(&consumer, &m)
}
```

```
void consumer()
{
    recv(&producer, &m);
    unpack(&m, &item)
    send(&producer, &m);
    consume(item);
}
```

- Mailboxes are used as temporary storage space and must be large enough to hold all tokens / messages

Equivalence of Mechanisms

- Are there synchronization problems which can be solved only with a subset of the mechanisms?
- Or are all the mechanisms equivalent?
- Constructive proof technique:
  - Two mechanisms A and B are equivalent if A can emulate B and B can emulate A
  - In both proof directions, construct an emulation (does not have to be efficient - just correct ;-)

Synchronization in Java

- Java supports mutual exclusion of blocks by declaring them synchronized:

```
synchronized(expr) {
    // 'expr' must evaluate to an Object
}
```

- Java supports mutual exclusion to critical sections of an object by marking methods as `synchronized`, which is in fact just syntactic sugar:

```
synchronized void foo()  { /* body */ }
void foo() { synchronized(this) { /* body */ } }
```

- Additional `wait()`, `notify()` and `notifyAll()` methods can be used to coordinate critical sections

Bounded Buffer in Java

```
class BoundedBuffer
{
    private final int size = 8;
    private int count = 0, out = 0, in = 0;
    private int[] buffer = new int[size];

    public synchronized void insert(int i)
    {
        while (count == size) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        buffer[in] = i;
        in = (in + 1) % size;
        count++;
        notifyAll();    // wakeup all waiting threads
    }
```

Bounded Buffer in Java

```
    public synchronized int remove(int i)
    {
        while (count == 0) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        int r = buffer[out];
        out = (out + 1) % size;
        count--;
        notifyAll();    // wakeup all waiting threads
        return r;
    }
}
```

- Java locks are reentrant (a thread cannot lock on itself)

Synchronization in Go

- Light-weigth "goroutines" that are typically mapped to an operating system level thread pool
- Channels provide message queues between goroutines
- Philosophy: Do not communicate by sharing memory; instead, share memory by communicating
- Inspired by Hoares work on Communicating Sequential Processes (CSP)

POSIX Mutex Locks

```
#include <pthread.h>

typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;

int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abstime);
```

- Mutex locks are a simple mechanism to achieve mutual exclusion in critical sections

POSIX Condition Variables

```
#include <pthread.h>

typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *condattr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           struct timespec *abstime);
```

- Condition variables can be used to bind the entrance into a critical section protected by a mutex to a condition

POSIX Barriers

```
#include <pthread.h>

typedef ... pthread_barrier_t;
typedef ... pthread_barrierattr_t;

int pthread_barrier_init(pthread_barrier_t *barrier,
                         pthread_barrierattr_t *barrierattr,
                         unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- Barriers block threads until the required number of threads have called `pthread_barrier_wait()`.

POSIX Message Queues

```
#include <mqueue.h>

typedef ... mqd_t;

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
               struct mq_attr *oldattr);

int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
```

- Message queues can be used to exchange messages between threads and processes running on the same system efficiently

POSIX Message Queues

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned int msg_prio);
int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
                 size_t msg_len, unsigned int msg_prio,
                 const struct timespec *abs_timeout);

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio);
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr,
                        size_t msg_len, unsigned int *msg_prio,
                        const struct timespec *abs_timeout);

int mq_notify(mqd_t mqdes, const struct sigevent *sevp);
```

- Message queues notifications can be delivered in different ways, e.g., as signals or in a thread-like fashion

POSIX Semaphores

```
#include <semaphore.h>

typedef ... sem_t;

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);

sem_t* sem_open(const char *name, int oflag);
sem_t* sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

- Unnamed semaphores are created with (`sem_init()`)
- Named semaphores are created with (`sem_open()`)

Atomic Operations in Linux (2.6.x)

```
struct ... atomic_t;

int  atomic_read(atomic_t *v);
void atomic_set(atomic_t *v, int i);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);

int atomic_add_negative(int i, atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_inc_and_test(atomic_t *v)
int atomic_dec_and_test(atomic_t *v);
```

- The `atomic_t` is essentially 24 bit wide since some processors use the remaining 8 bits of a 32 bit word for locking purposes

Atomic Operations in Linux (2.6.x)

```
void set_bit(int nr, unsigned long *addr);
void clear_bit(int nr, unsigned long *addr);
void change_bit(int nr, unsigned long *addr);

int  test_and_set_bit(int nr, unsigned long *addr);
int  test_and_clear_bit(int nr, unsigned long *addr);
int  test_and_change_bit(int nr, unsigned long *addr);
int  test_bit(int nr, unsigned long *addr);
```

- The kernel provides similar bit operations that are not atomic (prefixed with two underscores)
- The bit operations are the only portable way to set bits
- On some processors, the non-atomic versions might be faster

Spin Locks in Linux (2.6.x)

```
typedef ... spinlock_t;

void spin_lock(spinlock_t *l);
void spin_unlock(spinlock_t *l);
void spin_unlock_wait(spinlock_t *l);
void spin_lock_init(spinlock_t *l);
int  spin_trylock(spinlock_t *l)
int  spin_is_locked(spinlock_t *l);

typedef ... rwlock_t;

void read_lock(rwlock_t *rw);
void read_unlock(rwlock_t *rw);
void write_lock(rwlock_t *rw);
void write_unlock(rwlock_t *rw);
void rwlock_init(rwlock_t *rw);
int  write_trylock(rwlock_t *rw);
int  rwlock_is_locked(rwlock_t *rw);
```

Semaphores in Linux (2.6.x)

```
struct ... semaphore;

void sema_init(struct semaphore *sem, int val);
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);

void down(struct semaphore *sem);
int  down_interruptible(struct semaphore *sem);
int  down_trylock(struct semaphore *sem);

void up(struct semaphore *sem);
```

- Linux kernel semaphores are counting semaphores
- `init_MUTEX(s)` equals `sema_init(s, 1)`
- `init_MUTEX_LOCKED(s)` equals `sema_init(s, 0)`

Deadlocks

```
semaphore s1 = 1, s2 = 1;

void p1()
{
    down(&s1);
    down(&s2);
    critical_section();
    up(&s2);
    up(&s1);
}
```

```
void p2()
{
    down(&s2);
    down(&s1);
    critical_section();
    up(&s1);
    up(&s2);
}
```

- Executing the functions `p1` and `p2` concurrently can lead to a deadlock when both processes have executed the first `down()` operation
- Deadlocks also occur if processes do not release semaphores/locks

Deadlocks

```
class A
{
    public synchronized a1(B b)
    {
        b.b2();
    }

    public synchronized a2(B b)
    {
    }
}
```

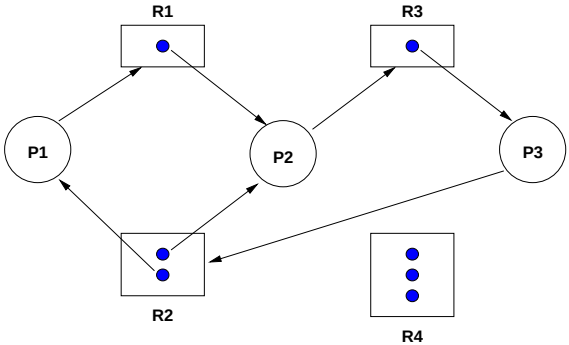
```
class B
{
    public synchronized b1(A a)
    {
        a.a2();
    }

    public synchronized b2(A a)
    {
    }
}
```

- Deadlocks can also be created by careless use of higher-level synchronization mechanisms
- Should the operating system not prevent deadlocks?

Necessary Deadlock Conditions

- *Mutual exclusion*:
  Resources cannot be used simultaneously by several processes
- *Hold and wait*:
  Processes apply for a resource while holding another resource
- *No preemption*:
  Resources cannot be preempted, only the process itself can release resources
- *Circular wait*:
  A circular list of processes exists where every process waits for the release of a resource held by the next process

Resource-Allocation Graph (RAG)



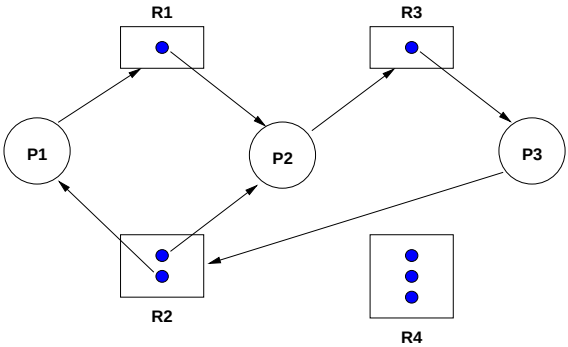
$$RAG = \{V, E\}$$
$$V = P \cup R$$
$$E = E_c \cup E_r \cup E_a$$

$P = \{P_1, P_2, \ldots, P_n\}$	(processes)
$R = \{R_1, R_2, \ldots, R_m\}$	(resource types)
$E_c = \{P_i \rightarrow R_j\}$	(resource claims (future))
$E_r = \{P_i \rightarrow R_j\}$	(resource requests (current))
$E_a = \{R_i \rightarrow P_i\}$	(resource assignments)

RAG Properties

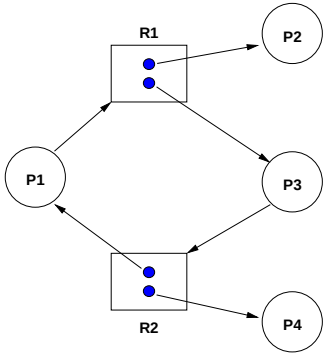
- Properties of a Resource-Allocation Graph:
  - A cycle in the RAG is a necessary condition for a deadlock
  - If each resource type has exactly one instance, then a cycle is also a sufficient condition for a deadlock
  - If each resource type has several instances, then a cycle is not a sufficient condition for a deadlock
- Dashed claim arrows (E_c) can express that a future claim for an instance of a resource is already known
- Information about future claims can help to avoid situations which can lead to deadlocks

RAG Example #1



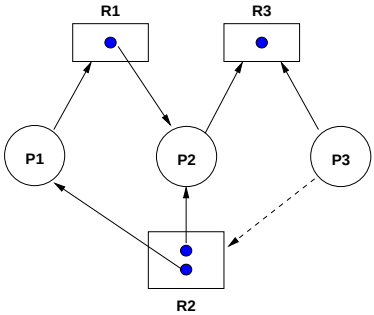
- Cycle 1: $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- Cycle 2: $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- Processes P_1, P_2 and P_3 are deadlocked

RAG Example #2



- Cycle: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- Processes P_1 and P_3 are not deadlocked
- P_4 may release its instance of R_2, breaking the cycle

RAG Example #3



- P_2 and P_3 both request R_3. To which process should the resource be assigned?
- Assign R_3 to P_2 to avoid a future deadlock situation

Deadlock Strategies

- *Prevention*:
  The system is designed such that deadlocks can never occur
- *Avoidance*:
  The system assigns resources so that deadlocks are avoided
- *Detection and recovery*:
  The system detects deadlocks and recovers itself
- *Ignorance*:
  The system does not care about deadlocks and the user has to take corrective actions

Deadlock Prevention

- Ensure that at least one of the necessary conditions cannot hold
- Prevent mutual exclusion:
  Some resources are intrinsically non-sharable
- Prevent hold and wait:
  Low resource utilization and starvation possible
- Prevent no preemption:
  Preemption can not be applied to some resources such as printers or tape drives
- Prevent circular wait:
  Leads to low resource utilization and starvation if the imposed order does not match process requirements

$\Longrightarrow$ Prevention is not feasible in the general case

Deadlock Avoidance

- Definitions:
  - A state is *safe* if the system can allocate resources to each process (up to its claimed maximum) and still avoid a deadlock
  - A state is *unsafe* if the system cannot prevent processes from requesting resources such that a deadlock occurs
- Assumption:
  - For every process, the maximum resource claims are known a priori.
- Idea:
  - Only grant resource requests that can not lead to a deadlock situation

Banker's Algorithm

- There are n processes and m resource types
- Let $i \in 1, \ldots, n$ and $j \in 1, \ldots m$
- $Total[j]$: total number of resources of type j
- $Avail[j]$: number of available resources of type j
- $Alloc[i, j]$: number of resources of type j allocated to process i
- $Max[i, j]$: maximum number of resources of type j claimed by process i to complete eventually
- $Need[i, j]$: number of requested resources of type j by process i

Banker's Algorithm

- Temporary variables:
  - $Work[j]$: available resources when some processes finish and deallocate
  - $Finish[i]$: boolean vector indicating processes able to finish
- Vector comparison:
  - Let X and Y be vectors of length n
  - $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, \ldots, n$

Safe-State Algorithm

1. Initialize:

   $Work \leftarrow Avail$
   $\forall i = 1, \ldots, n : Finish[i] \leftarrow false$

2. Select:

   Find a process i such that for $j = 1, \ldots, m$
   $Finish[i] = false \wedge Need[i, j] \leq Work[j]$
   If no such process i exists, go to step 4.

3. Update:

   $Work[j] \leftarrow Work[j] + Alloc[i, j]$ *for* $j = 1, \ldots, m$, $Finish[i] \leftarrow true$, *go to step 2.*

4. Finish:

   Safe state if $Finish[i] = true$ *for* $i = 1, \ldots, n$

Resource-Request Algorithm

1. Check:

   If $Request[j] \leq Need[j]$ *for* $j = 1, \ldots, m$, *go to step 2. Otherwise raise an error.*

2. Test:

   If $Request \leq Avail$, *go to step 3. Otherwise, process* i *must wait until resources are available*

3. Update:

   $$Avail[j] \leftarrow Avail[j] - Request[j]$$
   $$Alloc[i,j] \leftarrow Alloc[i,j] + Request[j]$$
   $$Need[i,j] \leftarrow Need[i,j] - Request[j]$$

4. Decide:

   If the resulting state is safe, the resource is allocated to process i. *Otherwise, process* i *must wait and the old state is restored*

Banker's Algorithm Example

- System description:

$$m = 4 \quad \text{resource types}$$

$$n = 5 \quad \text{processes}$$

$$Total = \{6, 8, 10, 12\}$$

$$Max = \begin{pmatrix} 3 & 1 & 2 & 5 \\ 3 & 2 & 5 & 7 \\ 2 & 6 & 3 & 1 \\ 5 & 4 & 9 & 2 \\ 1 & 3 & 8 & 9 \end{pmatrix}$$

Banker's Algorithm Example

- Can the system get into the state described by the following allocation matrix?

$$Alloc = \begin{pmatrix} 0 & 0 & 2 & 1 \\ 1 & 0 & 1 & 2 \\ 1 & 2 & 1 & 1 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 4 & 2 \end{pmatrix}$$

Banker's Algorithm Example

- Check whether the given state is safe:

$$Avail = (1, 2, 2, 6)$$

$$Need = \begin{pmatrix} 3 & 1 & 0 & 4 \\ 2 & 2 & 4 & 5 \\ 1 & 4 & 2 & 0 \\ 2 & 0 & 9 & 2 \\ 1 & 3 & 4 & 7 \end{pmatrix}$$

- The system may never reach this state!

Banker's Algorithm Example

- Assume the system is in the state described by the following matrix:

$$Alloc = \begin{pmatrix} 1 & 0 & 2 & 1 \\ 1 & 1 & 2 & 5 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 2 \end{pmatrix}$$

- How should the system react if process 4 requests an instance of resource 4?

Banker's Algorithm Example

- Assume the request can be granted:

$$Alloc = \begin{pmatrix} 1 & 0 & 2 & 1 \\ 1 & 1 & 2 & 5 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 0 & 2 & 2 \end{pmatrix} \quad Need = \begin{pmatrix} 2 & 1 & 0 & 4 \\ 2 & 1 & 3 & 2 \\ 1 & 4 & 0 & 0 \\ 4 & 3 & 8 & 0 \\ 0 & 3 & 6 & 7 \end{pmatrix}$$

- Is it still possible to satisfy the maximum claims?

Banker's Algorithm Example

- Maximum claims can be satisfied as shown below:

Avail	Action
$(1, 4, 0, 1)$	termination of process 3
$(2, 6, 3, 2)$	termination of process 2
$(3, 7, 5, 7)$	termination of process 1
$(4, 7, 7, 8)$	termination of process 5
$(5, 7, 9, 10)$	termination of process 4
$(6, 8, 10, 12)$	stop

- The new state is safe and the request can be granted.

Deadlock Detection

- Idea:
  - Assign resources without checking for unsafe states
  - Periodically run an algorithm to detect deadlocks
  - Once a deadlock has been detected, use an algorithm to recover from the deadlock
- Recovery:
  - Abort one or more deadlocked processes
  - Preempt resources until the deadlock cycle is broken
- Issues:
  - Criterias for selecting a victim?
  - How to avoid starvation?

Detection Algorithm

1. Initialize:

   $Work \leftarrow Avail$
   $\forall i = 1, \ldots, n : Finish[i] \leftarrow false$

2. Select:

   Find a process i such that for $j = 1, \ldots, m$
   $Finish[i] = false \wedge Request[i, j] \leq Work[j]$
   If no such process i exists, go to step 4.

3. Update:

   $Work[j] \leftarrow Work[j] + Alloc[i, j]$ *for* $j = 1, \ldots, m$, $Finish[i] \leftarrow true$, *go to step 2.*

4. Finish:

   Deadlock if $Finish[i] = false$ *for some* i, $1 \leq i \leq n$

CPU Scheduling

- A *scheduler* selects from among the processes in memory that are ready to execute, and allocates CPU to one of them.
- *Fairness*: Every process gets a fair amount of CPU time
- *Efficiency*: CPUs should be busy whenever there is a process ready to run
- *Response Time*: The response time for interactive applications should be minimized
- *Wait Time*: The time it takes to execute a given process should be minimized
- *Throughput*: The number of processes completed per time interval should be maximized

Preemptive Scheduling

- A *preemptive* scheduler can interrupt a running process and assign the CPU to another process
- A *non-preemptive* scheduler waits for the process to give up the CPU once the CPU has been assigned to the process

- Non-preemptive schedulers cannot guarantee fairness
- Preemptive schedulers are harder to design
- Preemptive schedulers might preempt the CPU at times where the preemption is costly (e.g., in the middle of a critical section)

Deterministic vs. Probabilistic

- A *deterministic* scheduler knows the execution times of the processes and optimizes the CPU assignment to optimize system behavior (e.g., maximize throughput)
- A *probabilistic* scheduler describes process behavior with certain probability distributions (e.g., process arrival rate distribution) and optimizes the overall system behavior based on these probabilistic assumptions

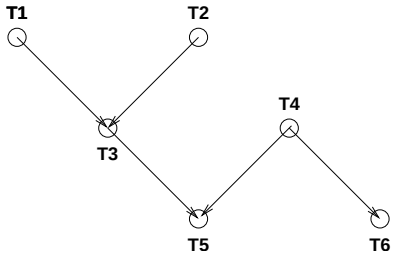
- Deterministic schedulers are relatively easy to analyze
- Finding optimal schedules is a complex problem
- Probabilistic schedulers must be analyzed using stochastic models (queuing models)

Deterministic Scheduling

- A *schedule* S for a set of processors $P = \{P_1, P_2, \ldots, P_m\}$ and a set of tasks $T = \{T_1, T_2, \ldots, T_n\}$ with the execution times $t = \{t_1, t_2, \ldots t_n\}$ and a set D of dependencies between tasks is a temporal assignment of the tasks to the processors.
- A *precedence graph* $G = (T, E)$ is a directed acyclic graph which defines dependencies between tasks. The vertices of the graph are the tasks T. An edge from T_i to T_j indicates that task T_j may not be started before task T_i is complete.

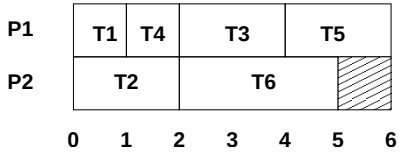
Deterministic Scheduling Example

- $T = \{T_1, T_2, T_3, T_4, T_5, T_6\}, n = 6$
- $t_1 = t_4 = 1, t_2 = t_3 = t_5 = 2, t_6 = 3$
- $G = (T, E)$
- $E = \{(T_1, T_3), (T_2, T_3), (T_3, T_5), (T_4, T_5), (T_4, T_6)\}$
- $P = \{P_1, P_2\}, m = 2$



Gantt Diagrams

- Schedules are often visualized using Gantt diagrams:



- Let $e = \{e_1, e_2, \ldots, e_n\}$ denote the termination time of the task $t_i \in T$ in the schedule S. The length of the schedule $t(S)$ and the average wait time $\overline{e}$ are defined as follows:

$$t(S) = \max_{1 \leq i \leq n} \{e_i\} \qquad \overline{e} = \frac{1}{n} \sum_{i=1}^{n} e_i$$

First-Come, First-Served (FCFS)

- Assumptions:
  - No preemption of running processes
  - Arrival and execution times of processes are known
- Principle:
  - Processors are assigned to processes on a first come first served basis (under observation of any precedences)
- Properties:
  - Straightforward to implement
  - Average wait time can become quite large

Longest Processing Time First (LPTF)

- Assumptions:
  - No preemption of running processes
  - Execution times of processes are known
- Principle:
  - Processors are assigned to processes with the longest execution time first
  - Shorter processes are kept to fill "gaps" later
- Properties:
  - For the length $t(S_L)$ of an LPTF schedule S_L and the length $t(S_O)$ of an optimal schedule S_O, the following holds:

$$t(S_L) \leq (\frac{4}{3} - \frac{1}{3m}) \cdot t(S_O)$$

Shortest Job First (SJF)

- Assumptions:
  - No preemption of running processes
  - Execution times of processes are known
- Principle:
  - Processors are assigned to processes with the shortest execution time first
- Properties:
  - The SJF algorithm produces schedules with the minimum average waiting time for a given set of processes and non-preemptive scheduling

Shortest Remaining Time First (SRTF)

- Assumptions:
  - Preemption of running processes
  - Execution times of the processes are known
- Principle:
  - Processors are assigned to processes with the shortest remaining execution time first
  - New arriving processes with a shorter execution time than the currently running processes will preempt running processes
- Properties:
  - The SRTF algorithm produces schedules with the minimum average waiting time for a given set of processes and preemptive scheduling

Round Robin (RR)

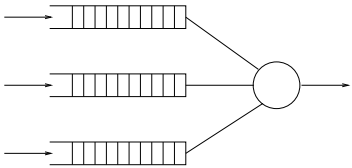
- Assumptions:
  - Preemption of running processes
  - Execution times or the processes are unknown
- Principle:
  - Processes are assigned to processors using a FCFS queue
  - After a small unit of time (time slice), the running processes are preempted and added to the end of the FCFS queue
- Properties:
  - time slice $\rightarrow \infty$: FCFS scheduling
  - time slice $\rightarrow 0$: processor sharing (idealistic)
  - Choosing a "good" time slice is important

Round Robin Variations

- Use separate queues for each processor
  - keep processes assigned to the same processor
- Use a short-term queue and a long-term queue
  - limit the number of processes that compete for the processor on a short time period
- Different time slices for different types of processes
  - degrade impact of processor-bound processes on interactive processes
- Adapt time slices dynamically
  - can improve response time for interactive processes

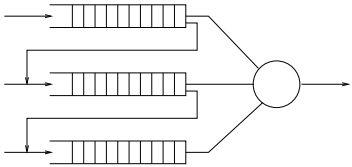
$\Longrightarrow$ Tradeoff between responsiveness and throughput

Multilevel Queue Scheduling



- Principle:
  - Multiple queues for processes with different priorities
  - Processes are permanently assigned to a queue
  - Each queue has its own scheduling algorithm
  - Additional scheduling between the queues necessary
- Properties:
  - Overall queue scheduling important (static vs. dynamic partitioning)

Multilevel Feedback Queue Scheduling



- Principle:
  - Multiple queues for processes with different priorities
  - Processes can move between queues
  - Each queue has its own scheduling algorithm
- Properties:
  - Very general and configurable scheduling algorithm
  - Queue up/down grade critical for overall performance

Real-time Scheduling

- *Hard real-time systems* must complete a critical task within a guaranteed amount of time
  - Scheduler needs to know exactly how long each operating system function takes to execute
  - Processes are only admitted if the completion of the process in time can be guaranteed
- *Soft real-time systems* require that critical tasks always receive priority over less critical tasks
  - Priority inversion can occur if high priority soft real-time processes have to wait for lower priority processes in the kernel
  - One solution is to give processes a high priority until they are done with the resource needed by the high priority process (priority inheritance)

Earliest Deadline First (EDF)

- Assumptions:
  - Deadlines for the real-time processes are known
  - Execution times of operating system functions are known
- Principle:
  - The process with the earliest deadline is always executed first
- Properties:
  - Scheduling algorithm for hard real-time systems
  - Can be implemented by assigning the highest priority to the process with the first deadline
  - If processes have the same deadline, other criterias can be considered to schedule the processes

Linux Scheduler System Calls (2.6.x)

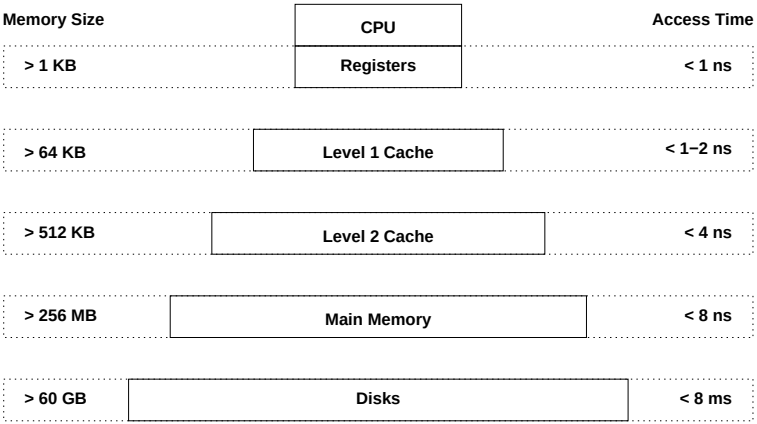
```
#include <unistd.h>

int nice(int inc);

#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);
int sched_getscheduler(pid_t pid);
int sched_setparam(pid_t pid, const struct sched_param *p);
int sched_getparam(pid_t pid, struct sched_param *p);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_rr_get_interval(pid_t pid, struct timespec *tp);
int sched_setaffinity(pid_t pid, unsigned int len, unsigned long *mask);
int sched_getaffinity(pid_t pid, unsigned int len, unsigned long *mask);
int sched_yield(void);
```

Parallel Computing Libraries

- OpenMP
  - Parallel computer API for shared memory systems. The OpenMP API is operating system independent and provides high-level compiler contructs (via C pragmas) for managing threads.
- OpenCL / OpenCL C
  - Parallel programming of heterogenous plattforms consisting of CPUs, GPUs, and DSPs. OpenCL provides access to GPUs for non graphical computing.
- Open MPI
  - A message passing library for distributed parallel computing based on the Message-Passing Interface standard (MPI)

Part: Memory Management

Memory Systems

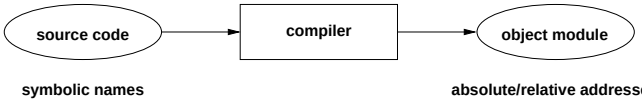


- In the following, we will focus on the main memory

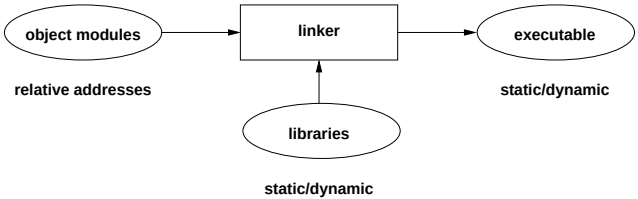
Main Memory

- Properties:
  - An ordered set of words or bytes
  - Each word or byte is accessible via a unique address
  - CPUs and I/O devices access the main memory
  - Running programs are (at least partially) loaded into main memory
  - CPUs usually can only access data in main memory directly (everything goes through main memory)
- Memory management of an operating system
  - allocates and releases memory regions
  - decides which process is loaded into main memory
  - controls and supervises main memory usage

Translation of Memory Addresses

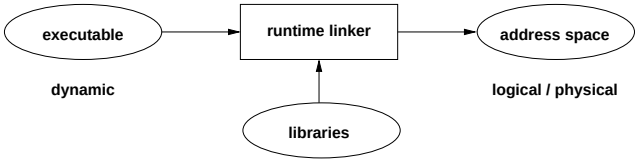


- Compiler translates symbolic addresses (variable / function names) into absolute or relative addresses

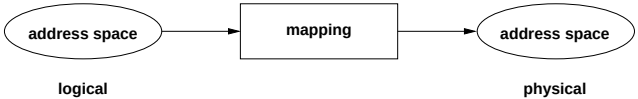


- Linker binds multiple object modules (with relative addresses) and referenced libraries into an executable

Translation of Memory Addresses



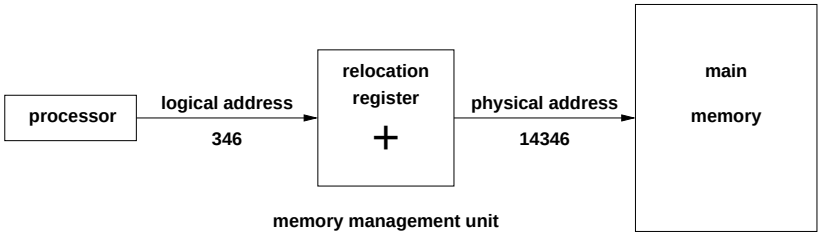
- Runtime linker binds executable with dynamic (shared) libraries at program startup time



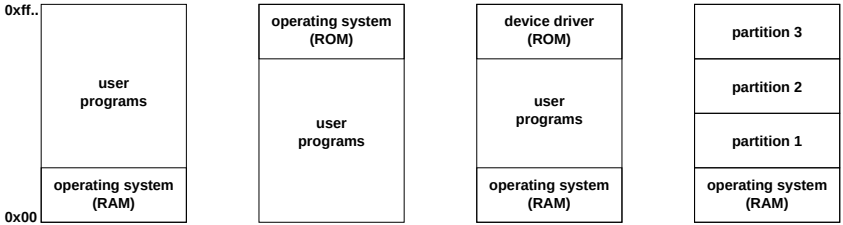
- Hardware memory management unit (MMU) maps the logical address space into the physical address space

Memory Management Tasks

- Dynamic memory allocation for processes
- Creation and maintenance of memory regions shared by multiple processes (shared memory)
- Protection against erroneous / unauthorized access
- Mapping of logical addresses to physical addresses

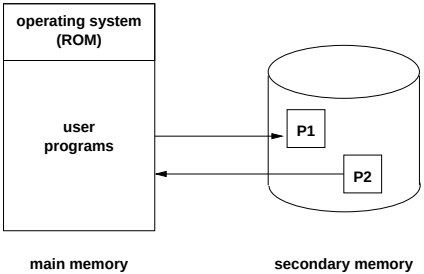


Memory Partitioning



- Memory space is often divided into several regions or partitions, some of them serve special purposes
- Partitioning enables the OS to hold multiple processes in memory (as long as they fit)
- Static partitioning is not very flexible (but might be good enough for embedded systems)

Swapping Principle

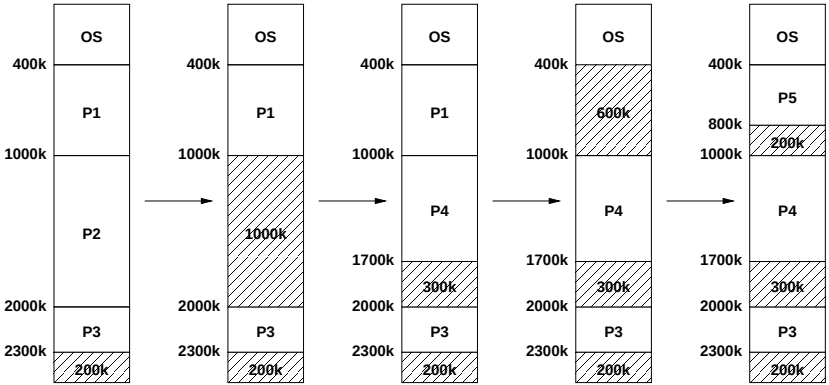


- Address space of a process is moved to a big (but slow) secondary storage system
- Swapped-out processes should not be considered runable by the scheduler
- Often used to handle (temporary) memory shortages

Segmentation

- Main memory is partitioned by the operating system into memory segments of variable length
  - Different segments can have different access rights
  - Segments may be shared between processes
  - Segments may grow or shrink
  - Applications may choose to only hold the currently required segments in memory (sometimes called overlays)
- Addition and removal of segments will over time lead to small unusable holes (external fragmentation)
- Positioning strategy for new segments influences efficiency and longer term behavior

External Fragmentation



- In the general case, there is more than one suitable hole to hold a new segment — which one to choose?

Positioning Strategies

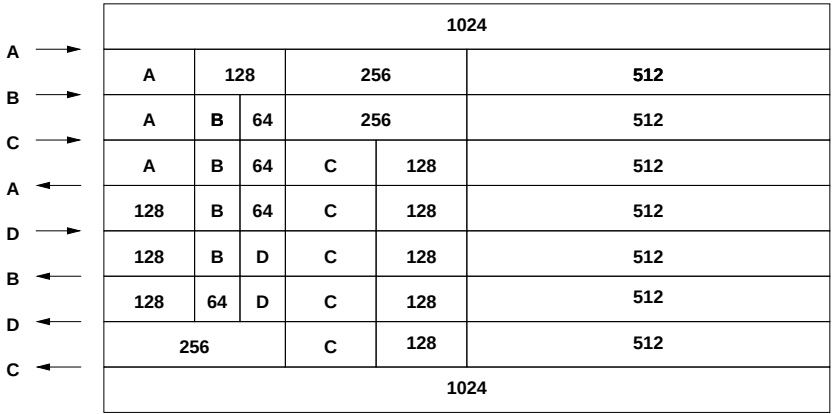
- *best fit*:
  - Allocate the smallest hole that is big enough
  - Large holes remain intact, many small holes
- *worst fit*:
  - Allocate the largest hole
  - Holes tend to become equal in size
- *first fit*:
  - Allocate the first hole from the top that is big enough
  - Simple and relatively efficient due to limited search
- *next fit*:
  - Allocate the next big enough hole from where the previous next fit search ended
  - Hole sizes are more evenly distributed

Positioning Strategies

- *buddy system*:
  - Holes always have a size of 2^i bytes (internal fragmentation)
  - Holes are maintained in k lists such that holes of size 2^i are maintained in list i
  - Holes in list i can be efficiently merged to a hole of size 2^{i+1} managed by list $i+1$
  - Holes in list i can be efficiently split into two holes of size 2^{i-1} managed by list $i-1$
  - Buddy systems are fast because only small lists have to be searched
  - Internal fragmentation can be costly
  - Used by user-space memory allocators (`malloc()`)

Buddy System Example

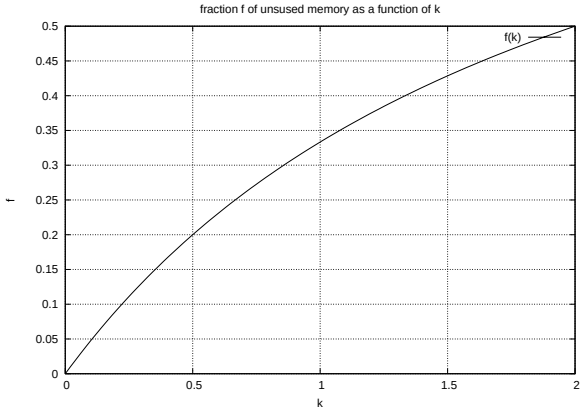
- Consider the processes A, B, C and D with the memory requests $70k$, $35k$, $80k$ and $60k$:



Segmentation Analysis

- *fifty percent rule*:
  Let n be the number of segments and h the number of holes. For large n and h and a system in equilibrium:
  $$h \approx \frac{n}{2}$$
- *unused memory rule*:
  Let s be the average segment size and ks the average hole size for some $k > 0$. With a total memory of m bytes, the $n/2$ holes occupy $m - ns$ bytes:
  $$(n/2) \cdot ks = m - ns \iff m = ns(1 + k/2)$$
  The fraction f of memory occupied by holes is:
  $$f = \frac{nks/2}{m} = \frac{nks/2}{ns(1 + k/2)} = \frac{k}{k + 2}$$

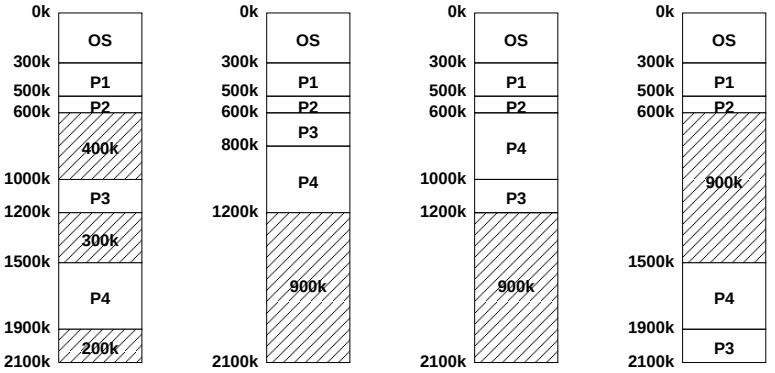
Segmentation Analysis



$\Longrightarrow$ As long as the average hole size is a significant fraction of the average process size, a substantial amount of memory will be wasted

Compaction

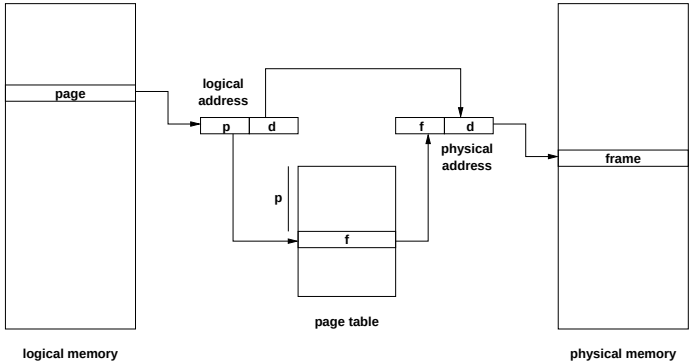
- Moving segments in memory allows to turn small holes into larger holes (and is usually quite expensive)
- Finding a good compaction strategy is not easy



Paging Idea

- General Idea:
  - Physical memory is organized in frames of fixed size
  - Logical memory is organized in pages of the same fixed size
  - Page numbers are mapped to frame numbers using a (very fast) page table mapping mechanism
  - Pages of a logical address space can be scattered over the physical memory
- Motivation:
  - Avoid external fragmentation and compaction
  - Allow fixed size pages to be moved into / out of physical memory

Paging Model and Hardware



- A logical address is a tuple (p, d) where p is an index into the page table and d is an offset within page p
- A physical address is a tuple (f, d) where f is the frame number and d is an offset within frame f

Paging Properties

- Address translation must be very fast (in some cases, multiple translations are necessary for a single machine instruction)
- Page tables can become quite large (a 32 bit address space with a page size of 4096 bytes requires a page table with 1 million entries)
- Additional information in the page table:
  - Protection bits (read/write/execute)
  - Dirty bit (set if page was modified)
- Not all pages of a logical address space must be resident in physical memory to execute the process
- Access to pages not in physical memory causes a page fault which must be handled by the operating system

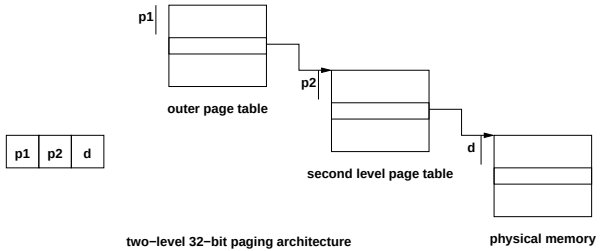
Handling Page Faults

1. MMU detects a page fault and raises an interrupt
2. Operating system saves the registers of the process
3. Mark the process blocked (waiting for page)
4. Determination of the address causing the page fault
5. Verify that the logical address usage is valid
6. Select a free frame (or a used frame if no free frame)
7. Write used frame to secondary storage (if modified)
8. Load page from secondary storage into the free frame
9. Update the page table in the MMU
10. Restore the instruction pointer and the registers
11. Mark the process runnable and call the scheduler

Paging Characteristics

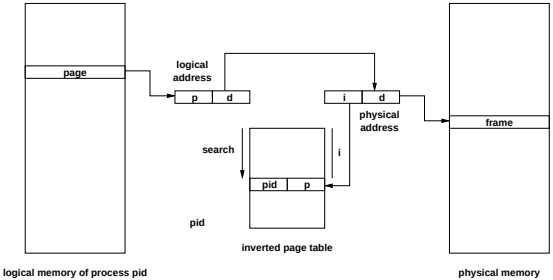
- Limited internal fragmentation (last page)
- Page faults are costly due to slow I/O operations
- Try to ensure that the "essential" pages of a process are always in memory
- Try to select used frames (victims) which will not be used in the future
- During page faults, other processes can execute
- What happens if the other processes also cause page faults?
- In the extreme case, the system is busy swapping pages into memory and does not do any other useful work (thrashing)

Multilevel Paging



- Paging can be applied to page tables as well
- SPARC 32-bit architecture supports three-level paging
- Motorola 32-bit architecture (68030) supports four-level paging
- Caching essential to alleviate delays introduced by multiple memory lookups

Inverted Page Tables



- The inverted page table has one entry for each frame
- Page table size determined by size of physical memory
- Entries contain page address and process identification
- The non-inverted page table is stored in paged memory
- Lookups require to search the inverted page table

Combined Segmentation and Paging

- Segmentation and paging have different strengths and weaknesses
- Combined segmentation and paging allows to take advantage of the different strengths
- Some architectures supported paged segments or even paged segment tables
- MMUs supporting segmentation and paging leave it to the operating systems designer to decide which strategy is used
- Note that fancy memory management schemes do not work for real-time systems...

Virtual Memory

- Virtual memory is a technique that allows the execution of processes that may not fit completely in memory
- Motivation:
  - Support virtual address spaces that are much larger than the physical address space available
  - Programmers are less bound by memory constraints
  - Only small portions of an address space are typically used at runtime
  - More programs can be in memory if only the essential data resides in memory
  - Faster context switches if resident data is small
- Most virtual memory systems are based on paging, but virtual memory systems based on segmentation are feasible

Loading Strategies

- Loading strategies determine when pages are loaded into memory:
  - *swapping*:
    Load complete address spaces (does not work for virtual memory)
  - *demand paging*:
    Load pages when they are accessed the first time
  - *pre-paging*:
    Load pages likely to be accessed in the future
  - *page clustering*:
    Load larger clusters of pages to optimize I/O
- Most systems use demand paging, sometimes combined with pre-paging

Replacement Strategies

- Replacement strategies determine which pages are moved to secondary storage in order to free frames
  - Local strategies assign a fixed number of frames to a process (page faults only affect the process itself)
  - Global strategies assign frames dynamically to all processes (page faults may affect other processes)
- Paging can be described using reference strings:

$w = r[1]r[2]\ldots r[t]\ldots$	sequence of page accesses
$r[t]$	page accessed at time t
$s = s[0]s[1]\ldots s[t]\ldots$	sequence of loaded pages
$s[t]$	set of pages loaded at time t
$x[t]$	pages paged in at time t
$y[t]$	pages paged out at time t

Replacement Strategies

- *First in first out (FIFO)*:
  Replace the page which is the longest time in memory
- *Second chance (SC)*:
  Like FIFO, except that pages are skipped which have been used since the last page fault
- *Least frequently used (LFU)*:
  Replace the page which has been used least frequently
- *Least recently used (LRU)*:
  Replace the page which has not been used for the longest period of time (in the past)
- *Belady's optimal algorithm (BO)*:
  Replace the page which will not be used for the longest period of time (in the future)

Belady's Anomaly

- Increasing memory size should decrease page fault rate
- Consider $w = 123412512345$, FIFO replacement strategy and the memory sizes $m = 3$ and $m = 4$:

```
s[0]  = {}                  s[0]  = {}
s[1]  = {1}     *           s[1]  = {1}       *
s[2]  = {1 2}   *           s[2]  = {1 2}     *
s[3]  = {1 2 3} *           s[3]  = {1 2 3}   *
s[4]  = {2 3 4} *           s[4]  = {1 2 3 4} *
s[5]  = {3 4 1} *           s[5]  = {1 2 3 4}
s[6]  = {4 1 2} *           s[6]  = {1 2 3 4}
s[7]  = {1 2 5} *           s[7]  = {2 3 4 5} *
s[8]  = {1 2 5}             s[8]  = {3 4 5 1} *
s[9]  = {1 2 5}             s[9]  = {4 5 1 2} *
s[10] = {2 5 3} *           s[10] = {5 1 2 3} *
s[11] = {5 3 4} *           s[11] = {1 2 3 4} *
s[12] = {5 3 4}             s[12] = {2 3 4 5} *
```

- 9 page faults for $m = 3$, 10 page faults for $m = 4$

Stack Algorithms

- Every reference string w can be associated with a sequence of stacks such that the pages in memory are represented by the first m elements of the stack
- A stack algorithm is a replacement algorithm with the following properties:
  1. The last used page is on the top
  2. Pages which are not used never move up
  3. Pages below the used page do not move
- Let $S_m(w)$ be the memory state reached by the reference string w and the memory size m
- For every stack algorithm, the following holds true:

$$S_m(w) \subseteq S_{m+1}(w)$$

LRU Algorithm

- LRU is a stack algorithm (while FIFO is not)
- LRU with counters:
  - CPU increments a counter for every memory access
  - Page table entries have a counter that is updated with the CPU's counter on every memory access
  - Page with the smallest counter is the LRU page
- LRU with a stack:
  - Keep a stack of page numbers
  - Whenever a page is used, move its page number on the top of the stack
  - Page number at the bottom identifies LRU page
- In general difficult to implement at CPU/MMU speed

Memory Management & Scheduling

- Interaction of memory management and scheduling:
  - Processes should not get the CPU if the probability for page faults is high
  - Processes must not remain in main memory if they are waiting for an event which is unlikely to occur in the near future
- How to estimate the probability of future page faults?
- Does the approach work for all programs equally well?
- Fairness?

Locality

- Locality describes the property of programs to use only a small subset of the memory pages during a certain part of the computation
- Programs are typically composed of several localities, which may overlap
- Reasons for locality:
  - Structured and object-oriented programming (functions, small loops, local variables)
  - Recursive programming (functional / declarative programs)
- Some applications (e.g., data bases or mathematical software handling large matrices) show only limited locality

Working-Set Model

- The *Working-Set* $W_p(t, T)$ of a process p at time t with parameter T is the set of pages which were accessed in the time interval $[t - T, t)$
- A memory management system follows the working-set model if the following conditions are satisfied:
  - Processes are only marked runnable if their full working-set is in main memory
  - Pages which belong to the working-set of a running process are not removed from memory
- Example ($T = 10$):

$$w = \ldots \underline{2615777751}623412\underline{344434344}413234443444\ldots$$

$$W(t_1) = \{1, 2, 5, 6, 7\} \qquad W(t_2) = \{3, 4\}$$

Working-Set Properties

- The performance of the working-set model depends on the parameter T:
  - If T is too small, many page faults are possible and thrashing can occur
  - If T is too big, unused pages might stay in memory and other processes might be prevented from becoming runnable
- Determination of the working-set:
  - Mark page table entries whenever they are used
  - Periodically read and reset these marker bits to estimate the working-set
- Adaptation of the parameter T:
  - Increase / decrease T depending on page fault rate

Part: Inter-Process Communication

Inter-Process Communication

- An operating system has to provide inter-process communication primitives in the form of system calls and APIs
- Signals:
  - Software equivalent of hardware interrupts
  - Signals interrupt the normal control flow, but they do not carry any data (except the signal number)
- Pipes:
  - Uni-directional channel between two processes
  - One process writes, the other process reads data
- Sockets:
  - General purpose communication endpoints
  - Multiple processes, global (Internet) communication

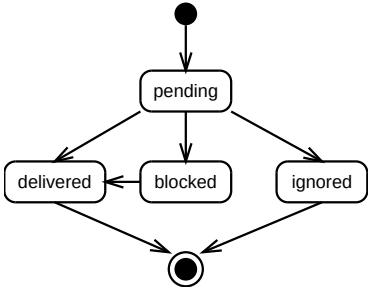
Signals

- Signals are a very basic IPC mechanism
- Basic signals are part of the standard C library
  - Signals for runtime exceptions (division by zero)
  - Signals created by external events
  - Signals explicitly created by the program itself
- Signals are either
  - *synchronous* or
  - *asynchronous* to the program execution
- POSIX signals are more general and powerful
- If in doubt, use POSIX signals to make code portable

C Library Signal API

```
#include <signal.h>

typedef ... sig_atomic_t;
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
int raise(int signum);

#define SIGABRT ...     /* abnormal  termination */
#define SIGFPE  ...     /* floating-point exception */
#define SIGILL  ...     /* illegal instruction */
#define SIGINT  ...     /* interactive interrupt */
#define SIGSEGV ...     /* segmentation violation */
#define SIGTERM ...     /* termination request */

#define SIG_IGN ...     /* handler to ignore the signal */
#define SIG_DFL ...     /* default handler for the signal */
#define SIG_ERR ...     /* handler returned on error situations */
```

POSIX Signal Delivery



- Signals start in the state *pending* and are usually *delivered* to the process quickly
- Signals can be *blocked* by processes
- Blocked signals are *delivered* when unblocked
- Signals can be ignored if they are not needed

Posix Signal API

```
#include <signal.h>

typedef void (*sighandler_t)(int);
typedef ... sigset_t;

#define SIG_DFL ...     /* default handler for the signal */
#define SIG_IGN ...     /* handler to ignore the signal */

#define SA_NOCLDSTOP ...
#define SA_ONSTACK   ...
#define SA_RESTART   ...

struct sigaction {
    sighandler_t sa_handler; /* handler function */
    sigset_t     sa_mask;    /* signals to block while executing handler */
    int          sa_flags;   /* flags to control behavior */
};
```

Posix Signal API

```
int sigaction(int signum, const struct sigaction *action,
              struct sigaction *oldaction);
int kill(pid_t pid, int signum);

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);

#define SIG_BLOCK   ...
#define SIG_UNBLOCK ...
#define SIG_SETMASK ...

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

Posix Signal API

- The function `sigaction()` registers a function to be executed when a specific signal has been received
- During the execution of a signal function, the triggering signal and any signals specified in the signal mask are blocked
- The function `kill()` sends a signal to a process or process group:
  - If `pid > 0`, the signal is sent to process pid.
  - If `pid == 0`, the signal is sent to every process in the process group of the current process
  - If `pid == -1`, the signal is sent to every process except for process 1 (init)
  - If `pid < -1`, the signal is sent to every process in the process group `-pid`

Properties of POSIX Signals

- Implementations can merge multiple identical signals
- Signals can not be counted reliably
- Signals do not carry any data / information except the signal number
- Signal functions are typically very short since the real processing of the signalled event is usually deferred to a later point in time of the execution when the state of the program is known to be consistent
- Variables modified by signals must be signal atomic
- `fork()` inherits signal functions, `exec()` resets signal functions (for security reasons)
- Threads in general share the signal actions, but every thread may have its own signal mask

Signal Example #1

```
#include <signal.h>

volatile sig_atomic_t keep_going = 1;

static void
catch_signal(int signum)
{
    keep_going = 0;          /* defer the handling of the signal */
}

int
main(void)
{
    signal(SIGINT, catch_signal);
    while (keep_going) {
        /* ... do something ... */
    }
    /* ... cleanup ... */
    return 0;
}
```

Signal Example #2

```
volatile sig_atomic_t fatal_error_in_progress = 0;

static void
fatal_error_signal(int signum)
{
    if (fatal_error_in_progress) {
        raise(signum);
        return;
    }
    fatal_error_in_progress = 1;
    /* ... cleanup ... */
    signal(signum, SIG_DFL);        /* install the default handler */
    raise(signum);                  /* and let it do its job */
}
```

- Template for catching fatal error signals
- Cleanup before raising the signal again with the default handler installed (which will terminate the process)

Signal Example #3

```
/*
 * sleep/sleep.c --
 *
 * This little example demonstrates how to use the POSIX signal
 * functions to wait reliably for a signal.
 */

#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static volatile sig_atomic_t wake_up = 0;

static void
catch_alarm(int sig)
{
    wake_up = 1;
}
```

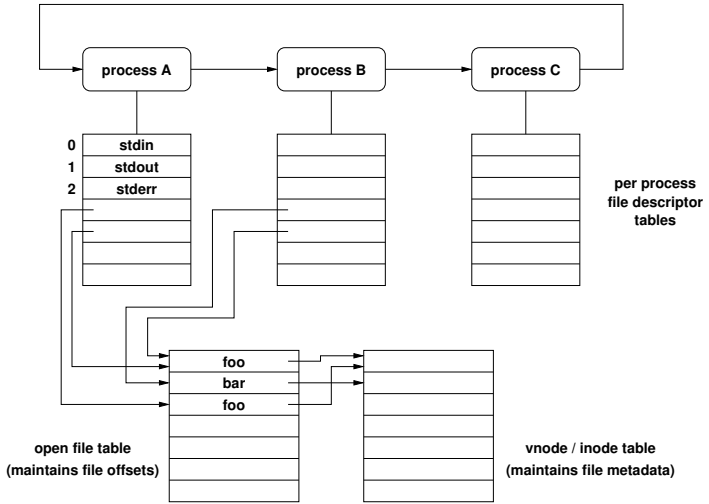
Signal Example #3 (cont.)

```
unsigned int
sleep(unsigned int seconds)
{
    struct sigaction sa, old_sa;
    sigset_t mask, old_mask;

    sa.sa_handler = catch_alarm;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    /* Be nice and save the original signal handler so that it can be
     * restored when we are done. */

    sigaction(SIGALRM, &sa, &old_sa);

    /* Ask the system to send us a SIGALRM at an appropriate time. */

    alarm(seconds);
```

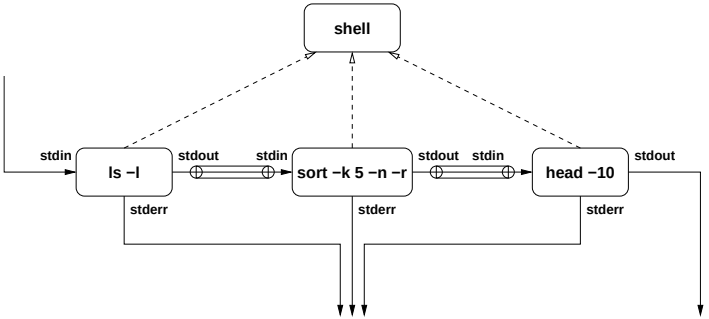
Signal Example #3 (cont.)

```
    /* First block the signal SIGALRM. After safely checking wake_up,
     * suspend until a signal arrives. Note that sigsuspend may return
     * on other signals. If wake_up is finally true, cleanup by
     * unblocking the blocked signals. */

    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigprocmask(SIG_BLOCK, &mask, &old_mask);

    /* No SIGALRM will be delievered here since it is blocked.
     * This means we have a safe region here until we suspend. */

    while (! wake_up) {
        /* Wait for SIGALRM (not blocked in old_mask). While
         * waiting, other signals may be handled as well ... */
        sigsuspend(&old_mask);
    }

    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    sigaction(SIGALRM, &old_sa, NULL);
    return 0;
}
```

Processes, File Descriptors, Open Files, . . .



Pipes at the Shell Command Line




```
# list the 10 largest files in the
# current directory
ls -l | sort -k 5 -n -r | head -10
```

POSIX Pipes

```
#include <unistd.h>

int pipe(int filedes[2]);
int dup(int oldfd);
int dup2(int oldfd, int newfd);

#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- Pipes can be used to send the output produced by one process as input to another process
- `popen()` and `pclose()` are wrappers to open a pipe to a child process executing the given command

Pipe Example: paging some text

```
static int
page(char *pager, char *text)
{
    ssize_t len, cnt;
    int status, pid, fd[2];

    status = pipe(fd);
    if (status == -1) {
        perror("pipe");
        return EXIT_FAILURE;
    }

    pid = fork();
    if (pid == -1) {
        perror("fork");
        return EXIT_FAILURE;
    }
```

Pipe Example

```
if (pid == 0) {
    close(fd[1]);
    status = dup2(fd[0], STDIN_FILENO);
    if (status == -1) {
        perror("dup2");
        return EXIT_FAILURE;
    }
    close(fd[0]);
    execl(pager, pager, NULL);
    perror("execl");
    _exit(EXIT_FAILURE);
} else {
    close(fd[0]);
    status = dup2(fd[1], STDOUT_FILENO);
    if (status == -1) {
        perror("dup2");
        return EXIT_FAILURE;
    }
```

Pipe Example

```
        close(fd[1]);
        for (len = strlen(text); len; len -= cnt, text += cnt) {
            cnt = write(STDOUT_FILENO, text, len);
            if (cnt == -1) {
                perror("write");
                return EXIT_FAILURE;
            }
        }
        close(1);
        do {
            if (waitpid(pid, &status, 0) == -1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
    }
    return EXIT_SUCCESS;
}
```

Named Pipes

- Pipes can only exist between processes which have a common parent process who created the pipe
- Named pipes are file system objects and arbitrary processes can read from or write to a named pipe
- Named pipes are created using the `mkfifo()` function
- A simple example:

```
$ mkfifo pipe
$ ls > pipe &
$ less < pipe
```

- An interesting example:

```
$ mkfifo pipe1 pipe2
$ echo -n x | cat - pipe1 > pipe2 &
$ cat < pipe2 > pipe1
```

Python Example

```
#!/usr/bin/env python

import os, sys

r, w = os.pipe()
pid = os.fork()
if pid:
    os.close(w)
    r = os.fdopen(r)   # turn r into a file object
    txt = r.read()
    os.waitpid(pid, 0) # make sure the child process gets cleaned up
else:
    os.close(r)
    w = os.fdopen(w, 'w')
    w.write("here's some text from the child")
    w.close()
    print "child: closing"
    sys.exit(0)

print "parent: got it; text =", txt
```

Sockets

- Sockets are abstract communication endpoints with a rather small number of associated function calls
- The socket API consists of
  - address formats for various network protocol families
  - functions to create, name, connect, destroy sockets
  - functions to send and receive data
  - functions to convert human readable names to addresses and vice versa
  - functions to multiplex I/O on several sockets
- Sockets are the de-facto standard communication API provided by operating systems

Socket Types

- Stream sockets (SOCK_STREAM) represent bidirectional reliable communication endpoints
- Datagram sockets (SOCK_DGRAM) represent bidirectional unreliable communication endpoints
- Raw sockets (SOCK_RAW) represent endpoints which can send/receive interface layer datagrams
- Reliable delivered message sockets (SOCK_RDM) are datagram sockets with reliable datagram delivery
- Sequenced packet sockets (SOCK_SEQPACKET) are stream sockets which retain data block boundaries

Generic Socket Addresses

```
#include <sys/socket.h>

struct sockaddr {
    uint8_t     sa_len          /* address length (BSD) */
    sa_family_t sa_family;      /* address family */
    char        sa_data[...];   /* data of some size */
};

struct sockaddr_storage {
    uint8_t     ss_len;         /* address length (BSD) */
    sa_family_t ss_family;      /* address family */
    char        padding[...];   /* padding of some size */
};
```

- A `struct sockaddr` represents an abstract address, typically casted to a struct for a concrete address format
- A `struct sockaddr_storage` provides storage space

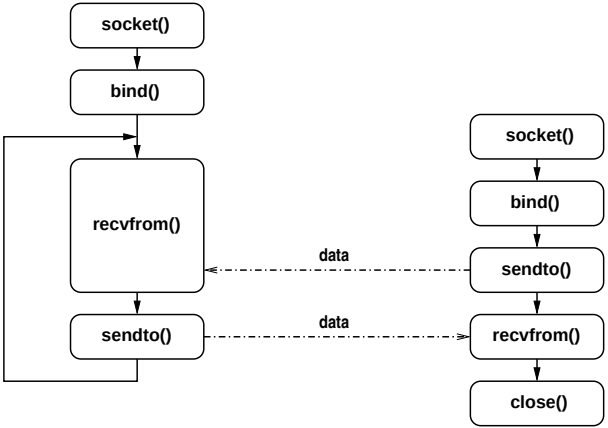
IPv4 Socket Addresses

```
#include <sys/socket.h>
#include <netinet/in.h>

typedef ... sa_family_t;
typedef ... in_port_t;

struct in_addr {
    uint8_t  s_addr[4];         /* IPv4 address */
};

struct sockaddr_in {
    uint8_t     sin_len;        /* address length (BSD) */
    sa_family_t sin_family;     /* address family */
    in_port_t   sin_port;       /* transport layer port */
    struct in_addr sin_addr;    /* IPv4 address */
};
```

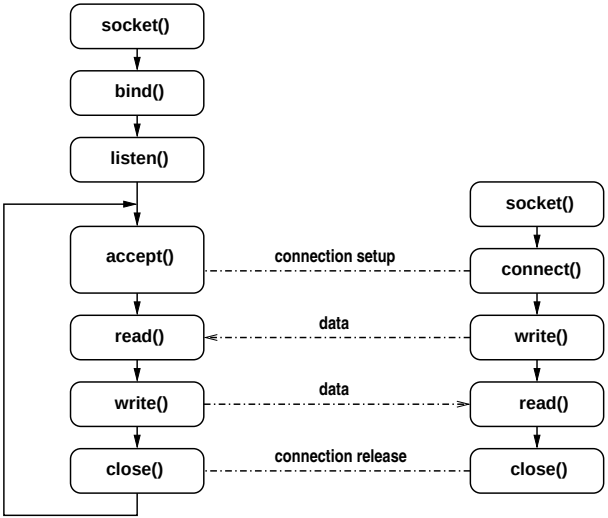
IPv6 Socket Addresses

```
#include <sys/socket.h>
#include <netinet/in.h>

typedef ... sa_family_t;
typedef ... in_port_t;

struct in6_addr {
    uint8_t  s6_addr[16];       /* IPv6 address */
};

struct sockaddr_in6 {
    uint8_t     sin6_len;       /* address length (BSD) */
    sa_family_t sin6_family;    /* address family */
    in_port_t   sin6_port;      /* transport layer port */
    uint32_t    sin6_flowinfo;  /* flow information */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t    sin6_scope_id;  /* scope identifier */
};
```

Connection-Less Communication



Connection-Oriented Communication



Socket API Summary

```
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

#define SOCK_STREAM    ...
#define SOCK_DGRAM     ...
#define SOCK_RAW       ...
#define SOCK_RDM       ...
#define SOCK_SEQPACKET ...

#define AF_LOCAL ...
#define AF_INET  ...
#define AF_INET6 ...

#define PF_LOCAL ...
#define PF_INET  ...
#define PF_INET6 ...
```

Socket API Summary

```
int socket(int domain, int type, int protocol);
int bind(int socket, struct sockaddr *addr,
         socklen_t addrlen);
int connect(int socket, struct sockaddr *addr,
            socklen_t addrlen);
int listen(int socket, int backlog);
int accept(int socket, struct sockaddr *addr,
           socklen_t *addrlen);

ssize_t write(int socket, void *buf, size_t count);
int send(int socket, void *msg, size_t len, int flags);
int sendto(int socket, void *msg, size_t len, int flags,
           struct sockaddr *addr, socklen_t addrlen);

ssize_t read(int socket, void *buf, size_t count);
int recv(int socket, void *buf, size_t len, int flags);
int recvfrom(int socket, void *buf, size_t len, int flags,
             struct sockaddr *addr, socklen_t *addrlen);
```

Socket API Summary

```
int shutdown(int socket, int how);
int close(int socket);

int getsockopt(int socket, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt(int socket, int level, int optname,
               void *optval, socklen_t optlen);
int getsockname(int socket, struct sockaddr *addr,
                socklen_t *addrlen);
int getpeername(int socket, struct sockaddr *addr,
                socklen_t *addrlen);
```

- All API functions operate on abstract socket addresses
- Not all functions make equally sense for all socket types

Mapping Names to Addresses

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define AI_PASSIVE     ...
#define AI_CANONNAME   ...
#define AI_NUMERICHOST ...

struct addrinfo {
    int             ai_flags;
    int             ai_family;
    int             ai_socktype;
    int             ai_protocol;
    size_t          ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

Mapping Names to Addresses

```
int getaddrinfo(const char *node,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
void freeaddrinfo(struct addrinfo *res);
const char *gai_strerror(int errcode);
```

- Many books still document the old name and address mapping functions
  - `gethostbyname()`
  - `gethostbyaddr()`
  - `getservbyname()`
  - `getservbyaddr()`

  which are IPv4 specific and should not be used anymore

Mapping Addresses to Names

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define NI_NOFQDN       ...
#define NI_NUMERICHOST  ...
#define NI_NAMEREQD     ...
#define NI_NUMERICSERV  ...
#define NI_NUMERICSCOPE ...
#define NI_DGRAM        ...

int getnameinfo(const struct sockaddr *sa,
                socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen,
                int flags);
const char *gai_strerror(int errcode);
```

Multiplexing

```
#include <sys/select.h>

typedef ... fd_set;

FD_ZERO(fd_set *set);
FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);

int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
int pselect(int n, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timespec *timeout,
            sigset_t sigmask);
```

- `select()` works with arbitrary file descriptors
- `select()` frequently used to implement the main loop of event-driven programs

Part: File Systems

File Types

- Files are persistent containers for the storage of data
- Unstructured files:
  - Container for a sequence of bytes
  - Applications interpret the contents of the byte sequence
  - File name extensions are often used to identify the type of contents (`.txt`, `.c`, `.pdf`)
- Structured files:
  - Sequential files
  - Index-sequential files
  - B-tree files

$\Longrightarrow$ Only some operating systems support structured files

Special Files

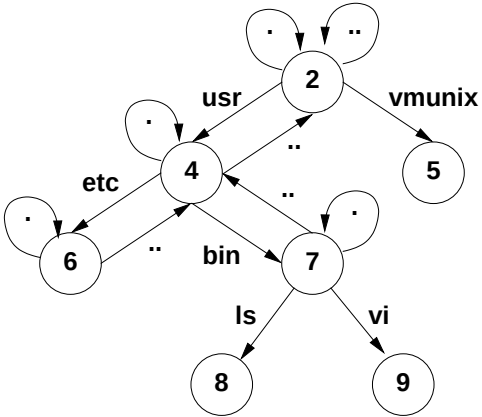
- Files representing devices:
  - Represent devices as files (`/dev/mouse`)
  - Distinction between block and character device files
  - Special operations to manipulate devices (`ioctl`)
- Files representing processes:
  - Represent processes (and more) as files (`/proc`)
  - Simple interface between kernel and system utilities
- Files representing communication endpoints:
  - Named pipes and fifos
  - Internet connection (`/net/tcp`) (Plan 9)
- Files representing graphical user interface windows:
  - Plan 9 represents all windows of a GUI as files

Directories

- Hierarchical name spaces
  - Files are the leaves of the hierarchy
  - Directories are the nodes spanning the hierarchy
- Names of files and directories on one level of the hierarchy usually have to be unique (beware of uppercase/lowercase and character sets)
- Absolute names formed through concatenation of directory and file names
- Directories may be realized
  - as special file system objects or
  - as regular files with special contents

$\Longrightarrow$ Small and embedded operating systems sometimes only support flat file name spaces

Unix Directory Structure

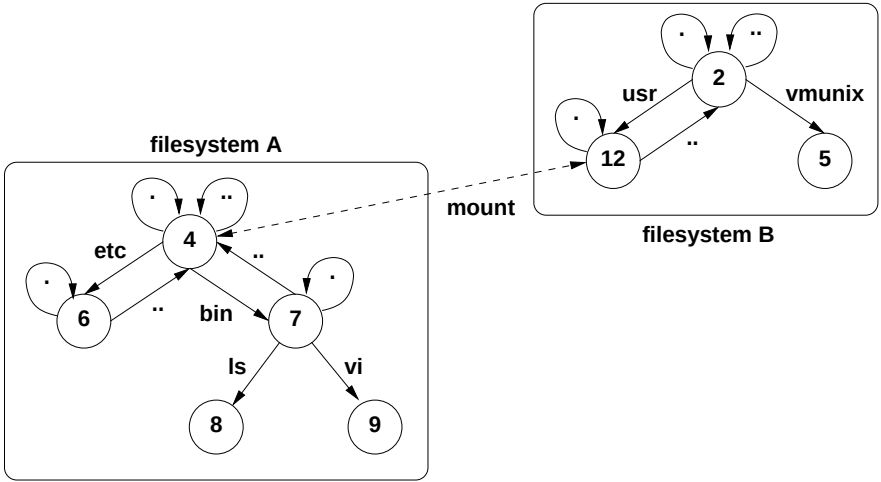


Mounting

- Mounting is the process of importing a directory (usually on some other media) into the existing name space
- Enables logical name spaces that span multiple devices
- Mounted file systems can be different
- Mounted directories may reside on remote systems

$\Longrightarrow$ More details on networked and distributed file systems in the Distributed Systems course

Mounting


filesystem A
.
..
4
etc
..
.
6
..
bin
7
.
ls
vi
8
9
mount
.
..
2
usr
vmunix
.
12
..
5
filesystem B

Links

- Access a single file or directory under different names
- Two common types of links:
  - Hard links register a file under two different names
  - Soft links store the path (pointer) of the real file
- Links usually turn hierarchical name spaces into directed graphs. What about cycles in the graph?

File Usage Pattern

- File usage patterns heavily depend on the applications and the environment
- Typical file usage pattern of "normal" users:
  - Many small files (less than 10K)
  - Reading is more dominant than writing
  - Access is most of the time sequential and not random
  - Most files are short lived
  - Sharing of files is relatively rare
  - Processes usually use only a few files
  - Distinct file classes
- Totally different usage patterns for e.g. databases

Standard File System Operations

- Most of the following C functions are C or POSIX standards

```
#include <stdlib.h>

int rename(const char *oldpath, const char *newpath);

#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
int close(int fd);
int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);
int access(const char *pathname, int mode);
int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
```

Standard File System Operations

```
#include <sys/types.h>
#include <sys/stat.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
int mkfifo(const char *pathname, mode_t mode);
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);

#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

Standard Directory Operations

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int chdir(const char *path);
int fchdir(int fd);

#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
int closedir(DIR *dir);
```

Memory Mapped Files

```
#include <sys/mman.h>

void* mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *start, size_t length);
int msync(void *start, size_t length, int flags);
int mprotect(const void *addr, size_t len, int prot);
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Direct mapping of regular files into virtual memory
- Enables extremely fast input/output and data sharing
- Mapped files can be protected and locked (regions)
- Changes made in memory are written to files during `unmap()` or `msync()` calls

Block Allocation Methods

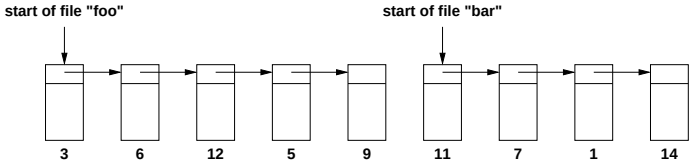
- *Contiguous allocation*:
  - Files stored as a contiguous block of data on the disk
  - Fast data transfers, simple to implement
  - File sizes often not known in advance
  - Fragmentation on disk
- *Linked list allocation*:
  - Every data block of a file contains a pointer (number) to the next data block
  - No fragmentation on disk
  - Reasonable sequential access, slow random access
  - Unnatural data block size (due to the space needed for the index)

Block Allocation Methods

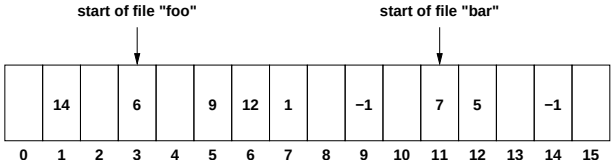
- *Linked list allocation using an index*:
  - The linked list is maintained in an index array outside of the data blocks
  - Index tables remain in main memory for fast access
  - Random access is reasonably faster
  - Significant memory usage by large index tables
  - Entire data blocks are available for data
- *Allocation using index nodes (inodes)*:
  - Small index nodes (inodes) store pointers to the first few disk blocks plus pointers to
    - an inode with data pointers (single indirect)
    - an inode with pointers to inodes (double indirect)
    - an inode with pointers to inodes with pointers to inodes (triple indirect)

Block Allocation Methods

- Linked list allocation example:

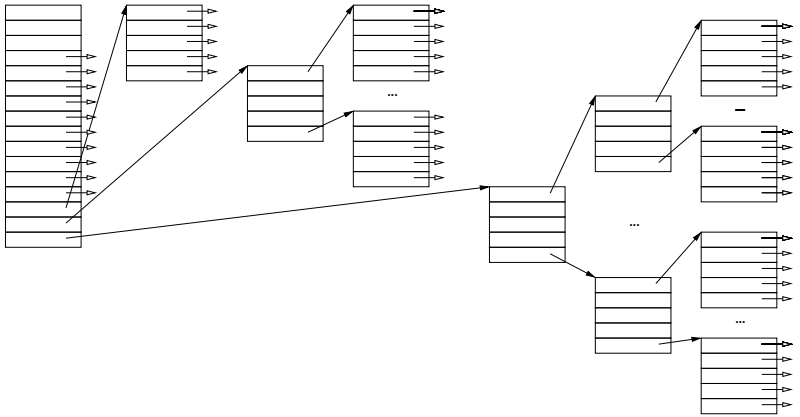


- Indexed linked list allocation example:



Block Allocation Methods

- Index node (inode) allocation example:



- Used on many Unix file systems (4.4 BSD and others)

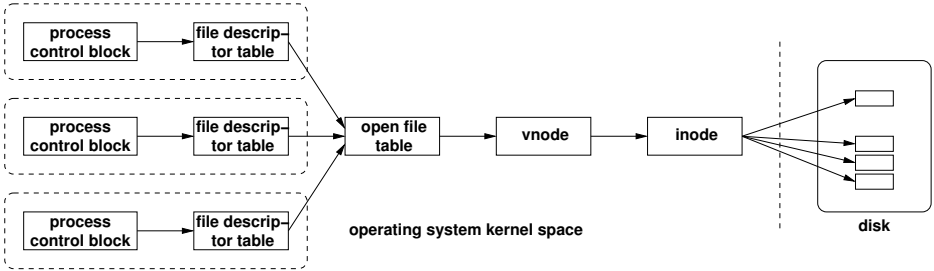
Free-Space Management

- *Free block lists*:
  - Manage free blocks in a linked free list
  - Efficient if there are only few free blocks
- *Free block bitmaps*:
  - Use a single bit for every block to indicate whether it is in use or not
  - Bitmap can be held in memory to make allocations and deallocations very fast
  - Sometimes useful to keep redundant bitmaps in order to recover from errors

Virtual File Systems (VFS)

- Provide an abstract (virtual) file system interface
- Common functions (e.g., caching) can be implemented on the virtual file system interface
- Simplifies support for many different file systems
- A virtual file system interface is often realized as a collection of function pointers
- Example Linux (`<linux/fs.h>`)
  - `struct super_operations`
  - `struct inode_operations`
  - `struct file_operations`

Processes and Files



- Every process control block maintains a pointer to the file descriptor table
- File descriptor table entries point to an entry in the open file table
- Open file table entries point to virtual inodes (vnodes)
- The vnode points to the inode (if it is a local file)

Part: Input/Output and Devices

Design Considerations

- Device Independence
  - User space applications should work with as many similar devices as possible without requiring any changes
  - Some user space applications may want to exploit specific device characteristics
  - Be as generic as possible while allowing applications to explore specific features of certain devices
- Efficiency
  - Efficiency is of great concern since many applications are I/O bound and not CPU bound
- Error Reporting
  - I/O operations have a high error probability and proper reporting of errors to applications and system administrators is crucial

Efficiency: Buffering Schemes

- Data is passed without any buffering from user space to the device (unbuffered I/O)
- Data is buffered in user space before it is passed to the device
- Data is buffered in user space and then again in kernel space before it is passed to the device
- Data is buffered multiple times in order to improve efficiency or to avoid side effects (e.g., flickering in graphics systems)
- Circular buffers can help to decouple data producer and data consumer without copying data
- Vectored I/O (scatter/gather I/O), uses a single function call to write data from multiple buffers to a single data stream or to read data from a data stream to multiple buffers

Efficiency: I/O Programming Styles

- *programmed input/output*:
  The CPU does everything (copying data to/from the I/O device) and blocks until I/O is complete
- *interrupt-driven input/output*:
  Interrupts drive the I/O process, the CPU can do other things while the device is busy
- *direct-memory-access input/output*:
  A DMA controller moves data in/out of memory and notifies the CPU when I/O is complete, the CPU does not need to process any interrupts during the I/O process

Error Reporting

- Provide a consistent and meaningful (!) way to report errors and exceptions to applications (and to system administrators)
- This is particularly important since I/O systems tend to be error prone compared to other parts of a computer
- On POSIX systems, system calls report errors via special return values and a (thread) global variable `errno` (`errno` stores the last error code and does not get cleared when a system call completes without an error)
- Runtime errors that do not relate to a specific system call are reported to a logging facility, usually via `syslog` on Unix systems

Representation of Devices

- Block devices represent devices where the natural unit of work is a fixed length data block (e.g., disks)
- Character devices represent devices where the natural unit of work is a character or a byte
- On Unix systems, devices are represented as special objects in the file system (usually mounted on `/dev`)
- Devices are identified by their type and their major and minor device number: the major number is used by the kernel to identify the responsible driver and the minor number to identify the device instance
- The `ioctl()` system call can be used by user-space applications to invoke device specific operations

Common Storage Media

- Magnetic disks (floppy disks, hard disks):
  - Data storage on rotating magnetic disks
  - Division into tracks, sectors and cylinders
  - Usually multiple (moving) read/write heads
- Solid state disks:
  - Data stored in solid-state memory (no moving parts)
  - Memory unit emulates hard disk interface
- Optical disks (CD, DVD, Blu-ray):
  - Read-only vs. recordable vs. rewritable
  - Very robust and relatively cheap
  - Division into tracks, sectors and cylinders
- Magnetic tapes (or tesa tapes):
  - Used mainly for backups and archival purposes
  - Not further considered in this lecture

RAID

- Redundant Array of Inexpensive Disks (1988)
- Observation:
  - CPU speed grows exponentially
  - Main memory sizes grow exponentially
  - I/O performance increases slowly
- Solution:
  - Use lots of cheap disks to replace expensive disks
  - Redundant information to handle high failure rate
- Common on almost all small to medium size file servers
- Can be implemented in hardware or software

RAID Level 0 (Striping)

- Striped disk array where the data is broken down into blocks and each block is written to a different disk drive
- I/O performance is greatly improved by spreading the I/O load across many channels and drives
- Best performance is achieved when data is striped across multiple controllers with only one drive per controller
- No parity calculation overhead is involved
- Very simple design
- Easy to implement
- Failure of just one drive will result in all data in an array being lost

RAID Level 1 (Mirroring)

- Twice the read transaction rate of single disks
- Same write transaction rate as single disks
- 100% redundancy of data means no rebuild is necessary in case of a disk failure
- Transfer rate per block is equal to that of a single disk
- Can sustain multiple simultaneous drive failures
- Simplest RAID storage subsystem design
- High disk overhead and thus relatively inefficient

RAID Level 2 (Striping + ECC)

- Write data to data disks
- Write error correcting codes (ECC) to ECC disks
- Read and correct data on the fly
- High data transfer rates possible
- The higher the data transfer rate required, the better the ratio of data disks to ECC disks
- Relatively simple controller design
- High ratio of ECC disks to data disks
- Entry level cost very high

RAID Level 3 (Striping + Parity)

- The data block is subdivided ("striped") and written on the data disks
- Stripe parity is generated on writes, recorded on the parity disk and checked on reads
- High read and write data transfer rate
- Low ratio of ECC (parity) disks to data disks
- Transaction rate equal to that of a single disk drive at best
- Controller design is fairly complex

RAID Level 4 (Parity)

- Data blocks are written onto data disks
- Parity for disk blocks is generated on writes and recorded on the shared parity disk
- Parity is checked on reads
- High read data transaction rate
- Low ratio of ECC (parity) disks to data disks
- Data can be restored if a single disk fails
- If two disks fail simultaneously, all data is lost
- Block read transfer rate equal to that of a single disk
- Controller design is fairly complex

RAID Level 5 (Distributed Parity)

- Data blocks are written onto data disks
- Parity for blocks is generated on writes and recorded in a distributed location
- Parity is checked on reads
- High read data transaction rate
- Data can be restored if a single disk fails
- If two disks fail simultaneously, all data is lost
- Block read transfer rate equal to that of a single disk
- Controller design is more complex
- Widely used in practice

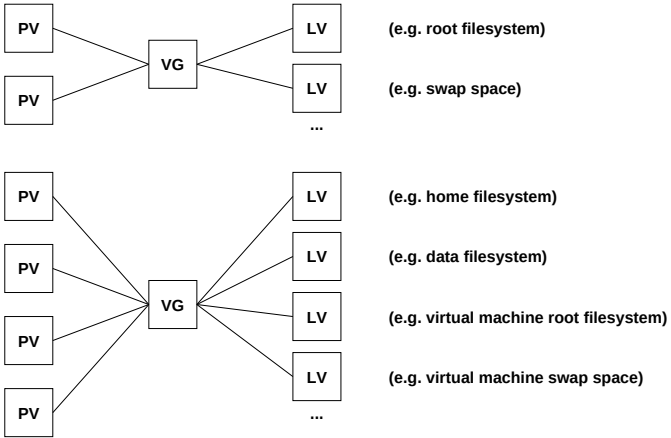
Logical Volume Management

- *Physical Volume*: A physical volume is a disk raw partition as seen by the operating system (hard disk partition, raid array, storage area network partition)
- *Volume Group*: A volume group pools several physical volumes into one logical unit
- *Logical Volume*: A logical volume resides in a volume group and provides a block device, which can be used to create a file system

$\Longrightarrow$ Separation of the logical storage layout from the physical storage layout

$\Longrightarrow$ Simplifies modification of logical volumes (create, remove, resize, snapshot)

Logical Volume Management (Linux)

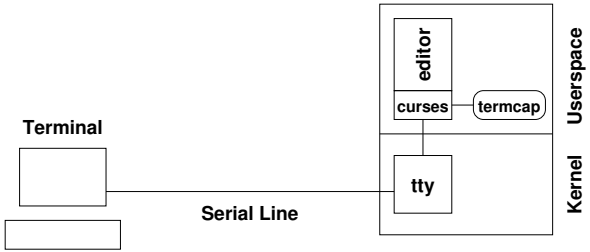


PV = physical volume, VG = volume group, LV = logical volume

Networked Storage

- Storage Area Networks (SAN)
  - A storage area network detaches block devices from computer systems through a fast communication network
  - Simplifies the sharing of storage between (frontend) computers
  - Dedicated network protocols (Fibre Channel, iSCSI, ...)
  - Relative expensive technology
- Network Attached Storage (NAS)
  - Access to a logical file system over the network
  - Sharing of file systems between multiple computers over a network
  - Many different protocols: NFS, SMB/CIFS, ...

Traditional Character Terminal Devices



- Character terminals were connected via serial lines
- The device driver in the kernel represents the terminal to user space programs (via a tty device file)
- Applications often use a library that knows about terminal capabilities to achieve terminal device independence

Serial Communication (RS232)

- Data transfer via two lines (TX/RX) using different voltage levels
- A *start bit* is used to indicate the beginning of the serial transmission of a word
- Parity bits may be sent (even or odd parity) to detect transmission errors
- One or several *stop bits* may be used after each word to allow the receiver to process the word
- *Flow control* can be implemented either using dedicated lines (RTS/CTS) or by sending special characters (XON/XOFF)
- Common settings: 8 data bits, 1 stop bit, no parity

Terminal Characteristics

- Serial lines were traditionally used to connect terminals to a computer
- Terminals understand different sets of control sequences (escape sequences) to control curser positioning or clearing of (parts of) the display
- Traditionally, terminals had different (often fixed) numbers of rows and columns they could display
- Keyboard were attached to the terminal and terminals did send different key codes, depending on the attached keyboard
- Teletypes were printers with an attached or builtin keyboard

Terminal Device

- Unix systems represent terminals as `tty` devices.
- In *raw mode*, no special processing is done and all characters received from the terminal are directly passed on to the application
- In *cooked mode*, the device driver preprocesses characters received from the terminal, generating signals for control character sequences and buffering input lines
- Terminal characteristics are described in the terminal capabilities (termcap, terminfo) databases
- The `TERM` variables of the process environment selects the terminal and thus the control sequences to send
- Network terminals use the same mechanism and are represented as pseudo tty devices called `ptys`.

Portable and Efficient Terminal Control

- Curses is a terminal control library enabling the construction of text user interface applications
- The curses API provides functions to position the cursor and to write at specific positions in a virtual window
- The refreshing of the virtual window to the terminal is program controlled
- Based on the terminal capabilities, the curses library can find the most efficient sequence of control codes to achieve the desired result
- The curses library also provides functions to switch between raw and cooked input mode and to control function key mappings
- The `ncurses` implementation provides a library to create panels, menus, and input forms.

Part: Virtual Machines

Virtualization Concepts in Operating Systems

- Virtualization has already been seen several times in operating system components:
  - virtual memory
  - virtual file systems
  - virtual block devices (LVM, RAID)
  - virtual terminal devices (pseudo ttys)
  - virtual network interfaces (not covered here)
  - ...
- What we are talking about now is running multiple operating systems on a single computer concurrently.

Emulation

- Emulation of processor architectures on different platforms
  - Transition between architectures (e.g., PPC $\Rightarrow$ Intel)
  - Faster development and testing of embedded software
  - Usage of software that can't be ported to new platforms
- Examples:
  - QEMU `http://www.qemu.org/`

Hardware Virtualization

- Virtualization of the physical hardware (aka hardware virtualization)
  - Running multiple operating systems concurrently
  - Consolidation (replacing multiple physical machines by a single machine)
  - Separation of concerns and improved robustness
  - High-availability (live migration, tandem systems, ...)
- Examples:
  - VMware — `http://www.vmware.com/`
  - VirtualBox — `https://www.virtualbox.org/`
  - Parallels — `http://www.parallels.com/`
  - Linux KVM — `http://www.linux-kvm.org/`
  - ...

User-Level Virtualization

- Virtualization of kernels in user space
  - Simplify kernel development and debugging
- Examples:
  - User-mode Linux `http://user-mode-linux.sourceforge.net/`

OS-Level Virtualization

- Multiple virtual operating system interfaces provided by a single operating system
  - Efficient separation using different namespaces
  - Robustness with minimal loss of performance
  - Reduction of system administration complexity
- Examples:
  - Linux Container
  - Linux VServer
  - BSD Jails
  - Solaris Zones

Paravirtualization

- Small virtual machine monitor controlling guest operating systems, relying on the help of guest operating systems
  - Efficient solution
  - Requiring OS support and/or hardware support
- Examples:
  - Xen `http://www.xenproject.org/`

Reading Material

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield.
Xen and the Art of Virtualization.
In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

Part: Distributed Systems

What is a Distributed System?

- A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. (Lesley Lamport, 1992)
- A distributed system is several computers doing something together. (M.D. Schroeder, 1993)
- An interconnected collection of autonomous computers, processes, or processors. (G. Tel, 2000)
- A distributed system is a collection of processors that do not share memory or a clock. (A. Silberschatz, 1994)

Why Distributed Systems?

- Information exchange
- Resource sharing
- Increased reliability through replication
- Increased performance through parallelization
- Simplification of design through specialization
- Cost reduction through open standards and interfaces

Challenges

General challenges for the design of distributed systems:

- Efficiency
- Scalability
- Security
- Fairness
- Robustness
- Transparency
- Openness

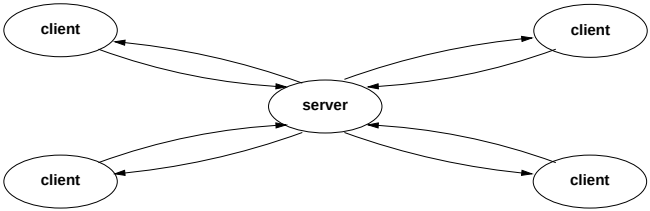
Special design challenges (increasingly important):

- Context-awareness and energy-awareness

Distributed vs. Centralized

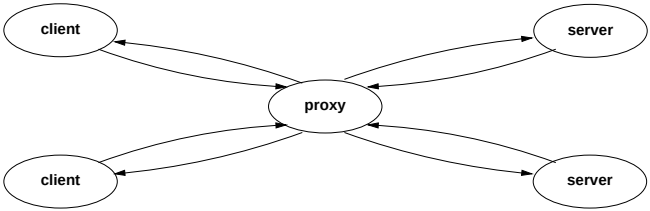
- *Lack of knowledge of global state*
  Nodes in a distributed system have access only to their own state and not to the global state of the entire system
- *Lack of a global time frame*
  The events constituting the execution of a centralized algorithm are totally ordered by their temporal occurance. Such a natural total order does not exist for distributed algorithms
- *Non-determinism*
  The execution of a distributed system is usually non-deterministic due to speed differences of system components

Client-Server Model



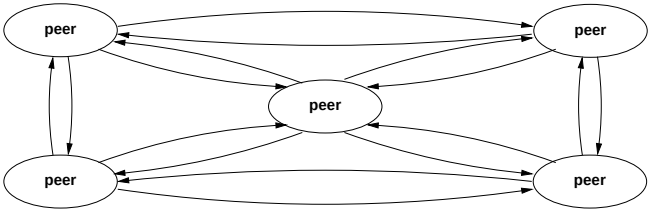
- Clients requests services from servers
- Synchronous: clients wait for the response before they proceed with their computation
- Asynchronous: clients proceed with computations while the response is returned by the server

Proxies



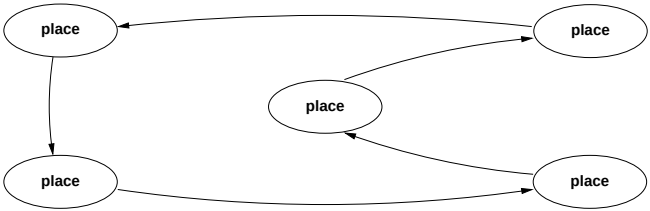
- Proxies can be introduced to
  - increase scalability
  - increase availability
  - increase protection and security
  - support multiple versions

Peer-to-Peer Model



- Every peer provides client and server functionality
- Avoids centralized components
- Able to establish new (overlay) topologies dynamically
- Requires control and coordination logic on each node

Mobile Code



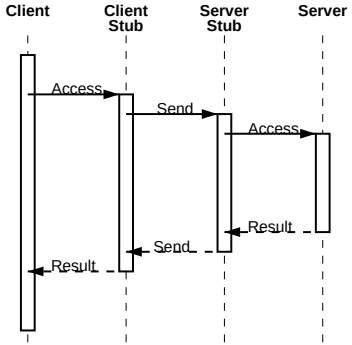
- Executable code (mobile agent) travels autonomously through the network
- At each place, some computations are performed locally that can change the state of the mobile agent
- Agent must be able to find a good trajectory
- Security (protection of places, protection of agents) is a difficult and serious problem

Taxonomy of Mobile Code

- *Applets*: downloadable applications
- *Servlets*: uploadable services
- *Extlets*: uploadable or downloadable features
- *Deglets*: delegated tasks
- *Netlets*: autonomous tasks
- *Piglets*: malicious mobile code

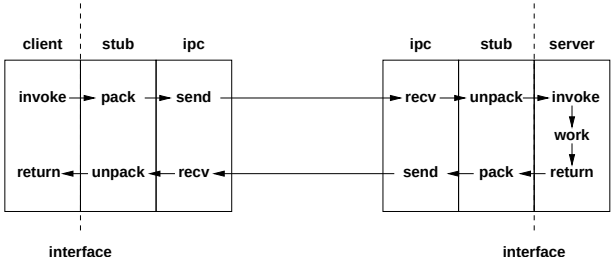
Andrzej Bieszczad, 1997

Remote Procedure Call Model



- Introduced by Birrel and Nelson (1984) to
  - provide communication transparency and
  - overcome heterogeneity

Stub Procedures

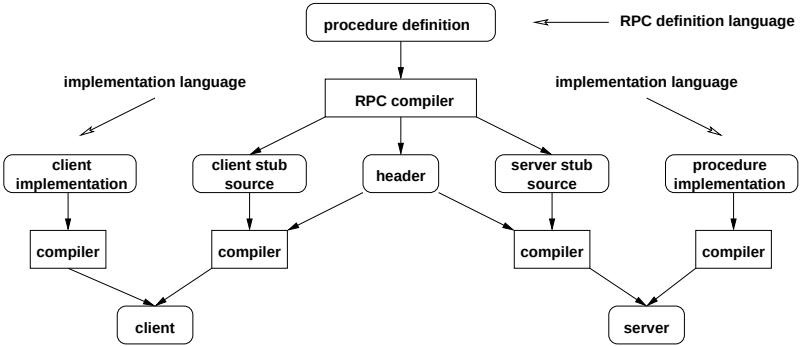


- Client stubs provide a local interface which can be called like any other local procedure
- Server stubs provide the server interface which calls the server's implementation of the procedure provided by a programmer and returns any results back to the client
- Stubs hide all communication details

Marshalling

- Marshalling is the technical term for transferring data structures used in remote procedure calls from one address space to another
- Serialization of data structures for transport in messages
- Conversion of data structures from the data representation of the calling process to that of the called process
- Pointers can be handled to some extend by introducing call-back handles which can be used to make an RPC call back from the server to the client in order to retrieve the data pointed to

RPC Definition Languages



- Formal language to define the type signature of remote procedures
- RPC compiler generates client / server stubs from the formal remote procedure definition

RPC Binding

- A client needs to locate and bind to a server in order to use RPCs
- This usually requires to lookup the transport endpoint for a suitable server in some sort of name server:
  1. The name server uses a well know transport address
  2. A server registers with the name server when it starts up
  3. A client first queries the name server to retrieve the transport address of the server
  4. Once the transport address is known, the client can send RPC messages to the correct transport endpoint

RPC Semantics

- *May-be*:
  - Client *does not* retry failed requests
- *At-least-once*:
  - Client *retries* failed requests, server re-executes the procedure
- *At-most-once*:
  - Client *may* retry failed requests, server detects retransmitted requests and responds with cached reply from the execution of the procedure
- *Exactly-once*:
  - Client *must* retry failed requests, server detects retransmitted requests and responds with cached reply from the execution of the procedure

Local vs. Remote Procedure Calls

- Client, server and the communication channel can fail independently and hence an RPC may fail
- Extra code must be present on the client side to handle RPC failures gracefully
- Global variables and pointers can not be used directly with RPCs
- Passing of functions as arguments is close to impossible
- The time needed to call remote procedures is orders of magnitude higher than the time needed for calling local procedures

Open Network Computing RPC

- Developed by Sun Microsystems (Sun RPC), originally published in 1987/1988
- Since 1995 controlled by the IETF (RFC 1790)
- ONC RPC encompasses:
  - ONC RPC Language (RFC 1831, RFC 1832)
  - ONC XDR Encoding (RFC 1832)
  - ONC RPC Protocol (RFC 1831)
  - ONC RPC Binding (RFC 1833)
- Foundation of the Network File System (NFS) and widely implemented on Unix systems

Distributed File Systems

- A *distributed file system* is a part of a distributed system that provides a user with a unified view of the files on the network
- Transparancy features (not necessarily all supported):
  - Location transparency
  - Access transparancy
  - Replication transparency
  - Failure transparency
  - Mobility transparency
  - Scaling transparency
- Recently: File sharing (copying) via peer-to-peer protocols

Design Issues

- Centralized vs. distributed data
  - Consistency of global file system state
  - If distributed, duplications (caching) or division
- Naming
  - Tree vs. Directed Acyclic Graph (DAG) vs. Forest
  - Symbolic links (file system pointers)
- File sharing semantics
  - Unix (updates are immediately visible)
  - Session (updates visible at end of session)
  - Transaction (updates are bundled into transactions)
  - Immutable (write once, change never)
- Stateless vs. stateful servers

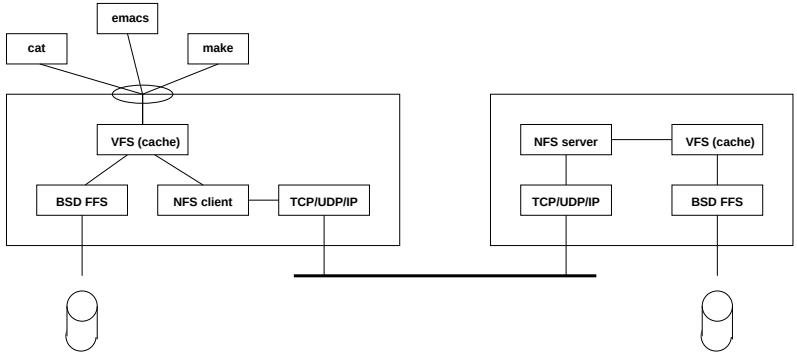
Stateless vs. Stateful Servers

- Stateless Server:
  - \+ Fault tolerance
  - \+ No open/close needed (less setup time)
  - \+ No data structures needed to keep state
  - \+ No limits on open files
  - \+ Client crashes do not impact the servers
- Stateful Server:
  - \+ Shorter request messages
  - \+ Better performance with buffering
  - \+ Readahead possible
  - \+ Idempotency is easier to achieve
  - \+ File locking possible

Network File System Version 3

- Original Idea:
  - Wrap the file system system calls into RPCs
  - Stateless server, little transparency support
  - Unix file system semantics
  - Simple and straight-forward to implement
  - Servers are dumb and clients are smart
- Stateless server
- Mount service for mounting/unmounting file systems
- Additional locking service (needs to be stateful)
- NFSv3 is defined in RFC 1813 (June 1995)

Operating System Integration



- Early implementations used user-space deamons
- NFS runs over UDP and TCP, currently TCP is preferred
- NFS uses a fixed port number (no portmapper involved)

NFSv3 Example (Simplified!)

```
C: PORTMAP GETPORT mount            # mount bayonne:/export/vol0 /mnt
S: PORTMAP GETPORT port
C: MOUNT /export/vol0
S: MOUNT FH=0x0222
C: PORTMAP GETPORT nfs              # dd if=/mnt/home/data bs=32k \
S: PORTMAP GETPORT port             # count=1 of=/dev/null
C: FSINFO FH=0x0222
S: FSINFO OK
C: GETATTR FH=0x0222
S: GETATTR OK
C: LOOKUP FH=0x0222 home
S: LOOKUP FH=0x0123
C: LOOKUP FH=0x0123 data
S: LOOKUP FH=0x4321
C: ACCESS FH=0x4321 read
S: ACCESS FH=0x4321 OK
C: READ FH=0x4321 at 0 for 32768
S: READ DATA (32768 bytes)
```

Related Work

- Distributed File Systems:
  - Network File System Version 4 (NFSv4) (2003)
  - Common Internet File System (CIFS) (2002)
  - Andrew File System (AFS) (1983)
  - ...
- Distributed File Sharing:
  - BitTorrent (2001)
  - Gnutella (2000)
  - Napster (1999)
  - ...

$\Longrightarrow$ Join the distributed systems course...