Modes

Modes:

- Kernel mode – all instructions allowed
- User mode – certain privileged instructions not allowed

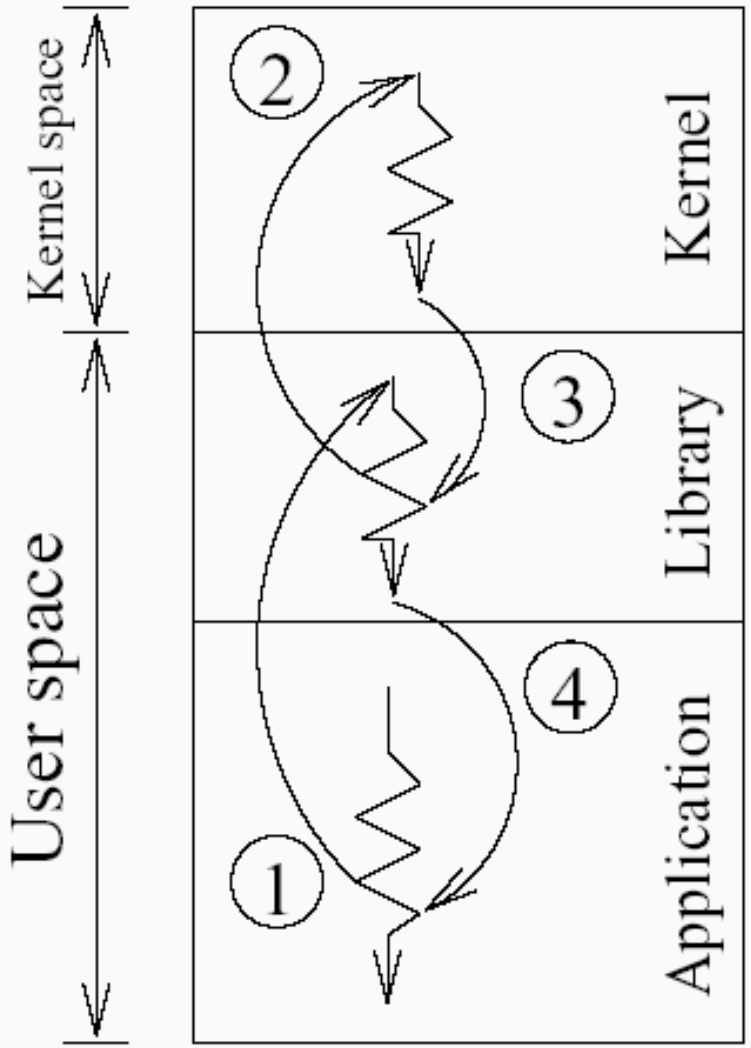
When do processes switch modes?

- System calls
- Exceptions
- Interrupts

Switching modes

1. **System calls** (traps/software interrupts)
   - synchronous events caused intentionally via machine instruction
   - serviced in *process context*
2. **Exceptions**
   - synchronous events caused by errors
     e.g. division by zero, accessing an illegal address
   - serviced in process context
3. **Interrupts**
   - asynchronous events caused by peripheral devices
   - serviced in *system context*
     - should not access process addess space or *u area*
     - should not block

System calls



1: Function call

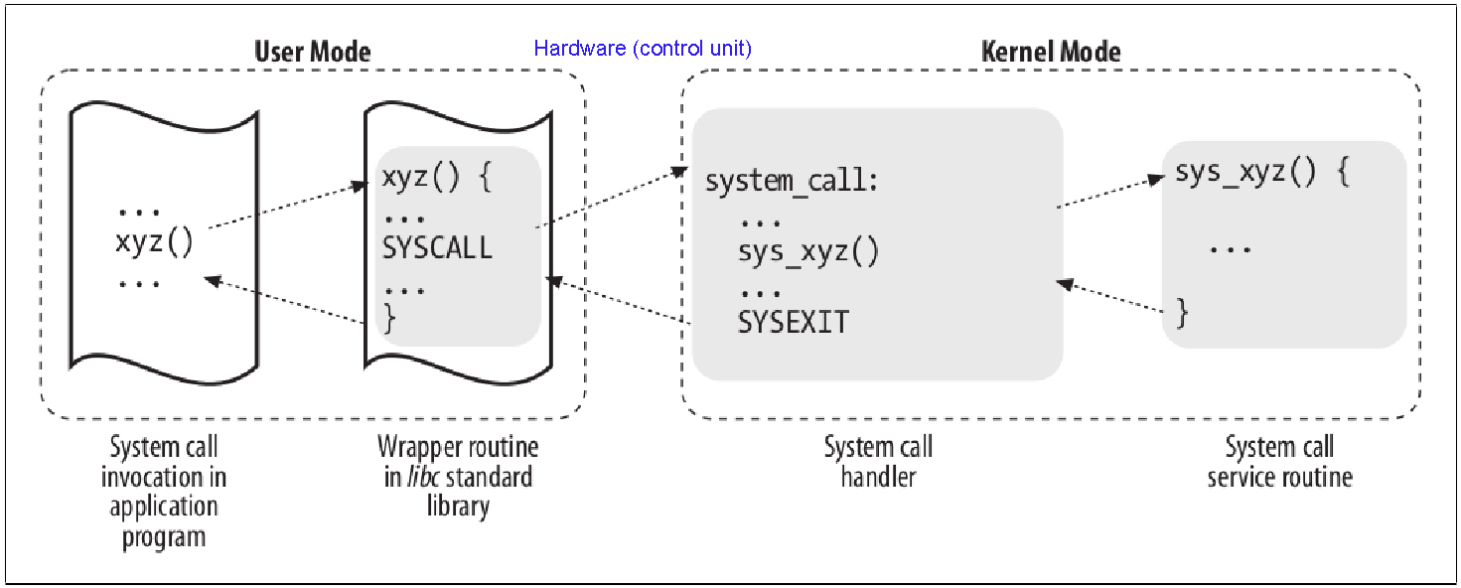
2: System call

3: System call return

4: Function call return

System calls in Linux

Reference: ULK 10, Bach 6.4.2



- `SYSCALL`, `SYSEXIT`: placeholders for actual assembly language instruction(s) to switch execution mode to kernel mode
  (generic name for this instruction: *trap* or *software interrupt*)
  - `int $0x80`, `iret` — traditional
  - `sysenter`, `sysexit` — "modern"

int instruction

- Determines the number ($\in \{0, \ldots, 255\}$) associated with the interrupt (e.g. `0x80`)
- Reads the corresponding entry of the Interrupt Descriptor Table (IDT)
  - pointed to by the `idtr` register
- Gets the address of the interrupt handler

System calls numbers

```
system_call:
pushl %eax # system call number is stored in register eax by
           # wrapper routine in libc
SAVE_ALL   # saves contents of (most) user registers in the kernel stack
```

- Kernel looks up system call number in a *dispatch vector* / *dispatch table* (`sys_call_table[NR_syscalls]` array) to find the address of the appropriate service routine

  `NR_syscalls` = 289 in the Linux 2.6.11 kernel

- Several library functions can map into one system call

Parameter passing and return value

- System calls cross from user to kernel mode
  $\Rightarrow$ neither stack can be used
  (working with two stacks at the same time is complex)
  $\Rightarrow$ parameters written in registers before issuing system call
- Size of each parameter cannot exceed the length of a register
  number of parameters must not exceed six
- Kernel copies parameters stored in the CPU registers onto kernel stack before invoking the system call service routine

Parameter passing and return value

- On return, kernel sets registers in the saved register context:
  - **on errors:**
    - sets carry bit in saved PSW
    - writes error number into a designated register in saved register context
  - **no errors:**
    - clears carry bit in saved PSW
    - copies return values from system call into a pair of designated registers
- When kernel returns to user mode, library function interprets return values from the kernel and returns suitable value to user program.

Definition

Reference: Section 4.1.1

Process: an executing instance of a program

Process vs. program:

- program is static (resides in file)
- many processes may correspond to the same program (e.g. `ls`, `pine`, etc.)

Kernel vs. processes

Reference: Vahalia 2.2

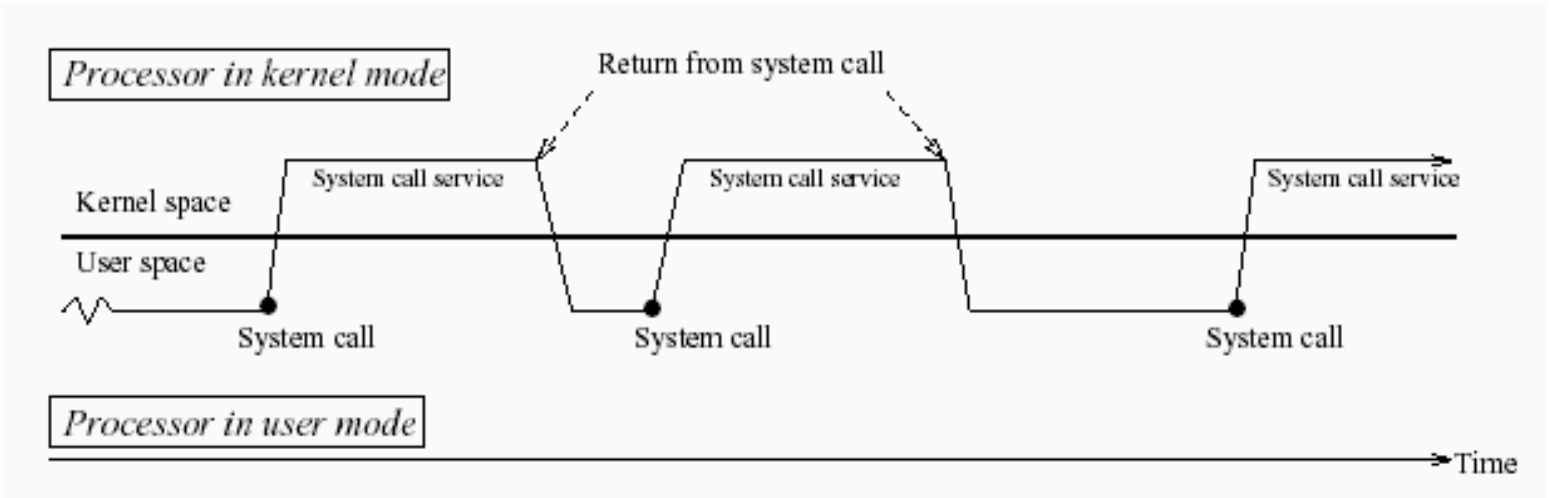
Kernel:

- code stored in `/vmunix`, `/boot/vmlinux`, etc.
- loaded into memory during booting

  (remains in memory until shutdown)
- initializes hardware and creates a few initial processes

Process:

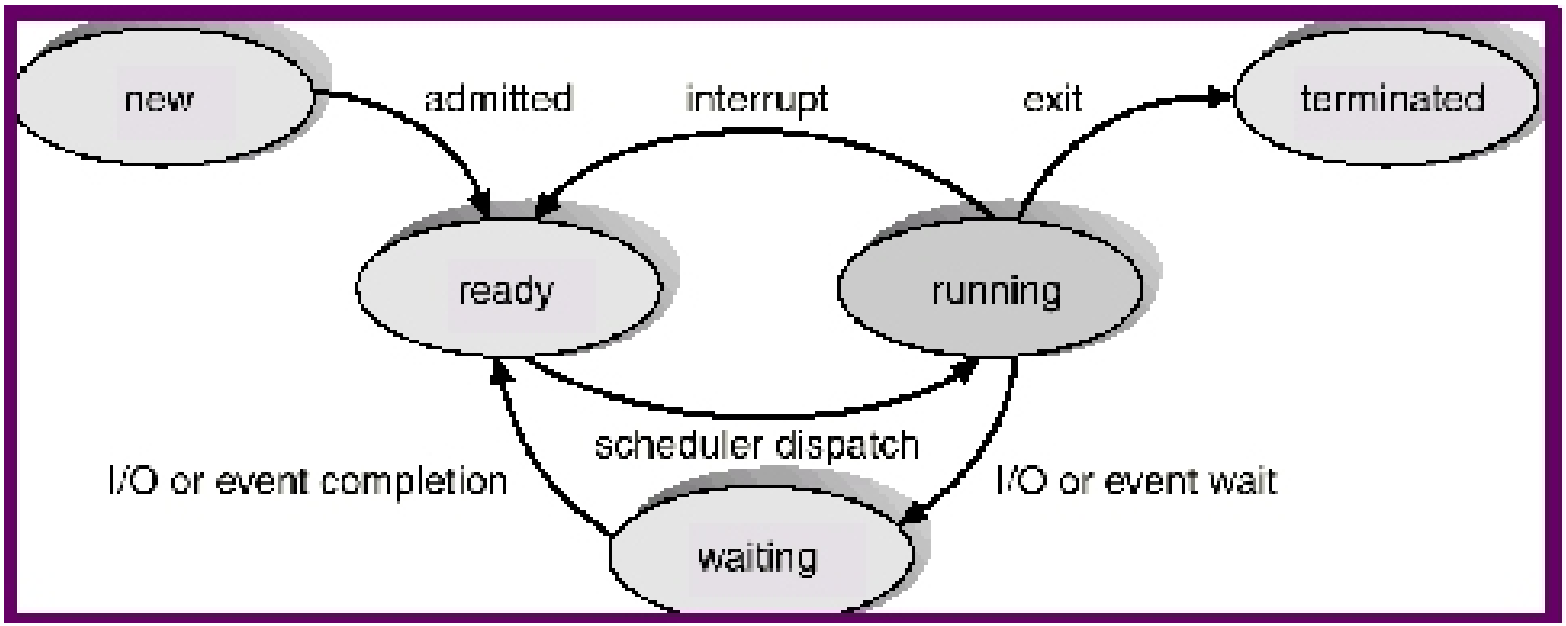
- makes calls to functions provided by the kernel in order to access hardware and other services



Process states

Reference: Section 4.1.2

- Typical process alternates between *computation* and *input/output*
- During I/O, CPU is idle
- For better utilization of resources, some other process should run during this time



Process context

Reference: Vahalia 2.3

Definition: “snapshot”, i.e. complete information about a process at some point during its execution

Constituents:

1. **User address space**
   - region of memory that the process can access (text, data, (user) stack, shared memory regions)
   - may be distributed through RAM / on-disk files / swap (special region of the disk)
2. **Registers**
   - general purpose registers, PC, SP, FPU registers
   - *processor status word* (PSW) – execution mode (current, previous), interrupt priority level (current, previous), overflow/carry bits
   - memory management registers

Process context

Constituents:

3. **Kernel stack**
   - stores activation records of kernel procedures when process is executing in kernel mode
   - empty when process is executing in user mode
4. **Address translation maps**
5. **Control information** – data structures used to store information about processes
   - *proc* structure – in kernel space (always visible to kernel)
   - *u area* – in process space (visible only for running process)
6. **Credentials** – user ID (UID) and group ID (GID)
7. **Environment variables**
   - set of strings of the form *VARIABLE=value*
   - usually stored at bottom of stack

Proc structure

1. Identification: process id, process group
2. Process state
3. Pointer to *u area*
4. Scheduling priority and related information
5. Memory management information (location in memory/storage)
6. Parent process id, pointers to parent, oldest child, immediate siblings
7. Pointers for linking process on hash queue (based on PID)
8. Pointers for linking process on scheduler / sleep queue
9. Signal information (masks of ignored, blocked, handled signals)

u area

1. Pointer to *proc* structure
2. Real and effective UID, GID
3. Process Control Block (***PCB***) – stores saved register context when process is not running
4. Size of text, data, stack regions
5. (Optional) Kernel stack for this process
6. Timing / usage information, disk quotas, resource limits
7. Arguments / return value from current system call
8. Table of open file descriptors
9. Pointer to current directory
10. Signal handlers

Credentials

- *Real* UID, GID: specified in `/etc/passwd`
- *Effective* UID, GID: determined by *suid* / *sgid* mode of the file containing the program

  e.g. `-r-s-x-x  1 root root  15104  Mar 14 2002 passwd`
- File creation, access: based on effective IDs
- Signalling: a process can send a signal to another only if the sender's real/effective UID matches the real UID of the receiver
- For superuser (*root*), UID = 0, GID = 1

⇐

Process switching

Reference: Bach 6.4.3

- Process switch can occur when a process
  1. puts itself to sleep (via *sleep*())
  2. exits
     (conclusion of exit system call invokes context switch code)
  3. returns from kernel mode to user mode but is not the most eligible process to run
- Kernel ensures consistency before switching context (completes necessary updates, does appropriate locking/unlocking operation) e.g.
  - when a process goes to sleep waiting for I/O, kernel locks allocated buffer so that no other process can access/corrupt it
  - link system call: releases lock of 1st inode before locking 2nd to prevent deadlock

Process switching

Principle:

1. Save the process context at some point.
2. Proceed to execute scheduling algorithm and context switch code in the context of the old process.
3. When context is restored later, execution should resume according to previously saved context.

Problem: distinguishing between 2 and 3

```
save_context(current);
/* scheduling algorithm */
resume_context(new);
```

```
if (save_context(current)) {
/* scheduling algorithm */
resume_context(new);
} /* resuming process starts here */
```

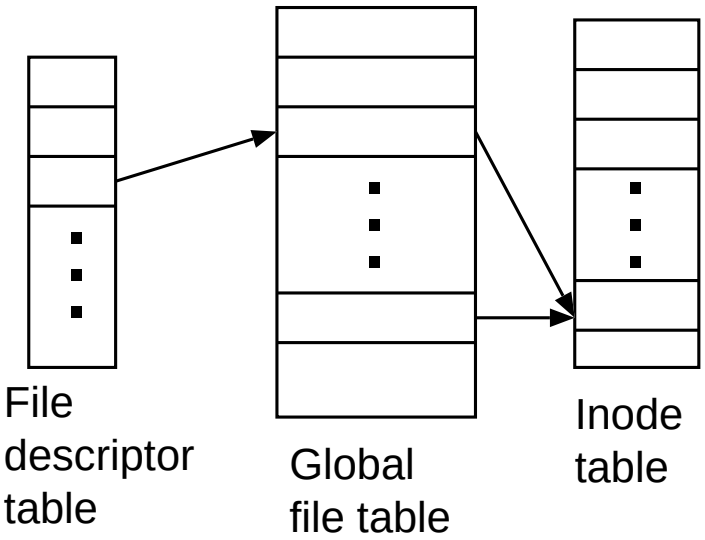
Process switching

1. Save current PC and other registers.
2. Set return value register of `save_context` to 0 in the saved register context.
3. Kernel continues to execute in the context of p_{old} to select p_{new}.
4. `resume_context` automatically switches to p_{new}.
5. When p_{old} is scheduled, PC is set to old value (saved in step 1).
6. Kernel resumes execution of p_{old} at the end of `save_context`.
7. On return, execution jumps over `resume_context` code.

ALT: PC may be set artificially to point to instruction where execution should resume.

Files

- File = header (*inode*) + data
- All files accessed through *inode*



- File descriptor (per process) – pointers to all open files
- Global file table – mode, offset for each `open`-ed file
- Inode table – memory copy of on-disk inode (only one per file)

Process creation

Reference: Bach 7.1, Vahalia 2.8.2

Syntax: `pid = fork();`

`pid` – PID of child process (parent)

`pid` – 0 (child)

Algorithm:

1. Check for available memory.
2. Check that user is not running too many processes.
3. Allocate new *proc* structure, assign new PID.

   (PIDS start from 1 and increase by 1 until wraparound at maximum value)
4. Copy data from parent *proc* structure to child.
   - real/effective UID, scheduling parameters, signal masks
   - parent process field of child is set, pointers to parent and sibling *procs*
   - child state is set to BEING CREATED

Process creation

5. Clear accounting information, timers, pending signals.
6. Connect new *proc* on relevant linked lists.
7. Increment reference count of inode of current directory.
   Increment global file table reference count associated with each open file of parent process.
   (child inherits access rights to open files
   child shares global file table entries with parent
   $\Rightarrow$ changes in file offset caused by read/write in the parent are visible to child and vice versa)
8. Allocate memory and create copy of parent context (*u area*, regions, page tables)
   (shared regions are not copied, only ref. count is incremented)
9. Copy parent's kernel-level context (registers + kernel stack).
10. Parent: return PID to user
    Child: "saved" context is restored, returns 0 to user

Process termination

Reference: Bach 7.3, Vahalia 2.8.5

Syntax: `exit(status);`

`status` – value returned to parent proc.

- may be called explicitly/implicitly (startup routine linked with all C programs calls exit when program returns from `main`)
- kernel may also invoke `exit()` when an uncaught signal is received

Algorithm:

1. Disable signal handling.
2. Close all open files, release inode for current directory.
3. Release all user memory.
4. Save exit status code and timing information in *proc*.
5. Write accounting record to file (UID, CPU/memory usage, amount of I/O, etc.)

Process termination

6. Change process state to ZOMBIE and put *proc* on zombie process list.
7. Assign parent PID of all live child processes to 1 (`init`);
   if any child process is ZOMBIE, current process sends `init` a SIGCHLD,
   `init` deletes *proc* structure for the process.
8. Send SIGCHLD to parent process.
9. Jump to context switch code.

Invoking a program I

Reference: Bach 7.5, Vahalia 2.8.4

Syntax: `execve(filename, argv, envp);`

`filename` – name of executable file

`argv` – parameters to program (`char **`)

`envp` – environment of program (`char **`)

Algorithm:

I. **Preliminary checks, preprocessing**

1. Check that file is an executable with proper permissions for the user.
2. Read file header to determine layout of the executable file.
   - primary header – magic number (specifies type of exec. file), no. of sections, start address for process execution
   - section headers – section type, size, virtual address occupied by the section
   - sections – code, data (initial contents of process address space)
   - misc. sections – symbol tables, debugging info, etc.

Invoking a program II

II. **Handle old address space.**

1. Copy parameters from old address space to kernel space.
   (old address space will be freed $\Rightarrow$ params have to be saved on:
   kernel stack + additional storage (if needed))
2. Free memory occupied by the process.

III. **Set up address space for new process.**

1. Allocate memory for the new process' code, data, stack.
2. Load contents of executable file into memory (code, initialized data).
3. Copy parameters to new user stack.
4. Set initial SP, PC (cf. file header).

Summary

- Relation between processes and the kernel (“operating system”)
- User mode vs. kernel mode, mode switches
  - System call algorithm
- Process states
  - processes alternate between running and waiting
  - for better CPU utilization, *multi-programming* is used
- Process context
  - needed in order to “freeze” and restart processes in a multiprogramming environment
  - any process $\equiv$ its *context* (complete information about the process at any point during its execution)
- Context switch algorithm
- `fork`, `exit`, *exec* system calls