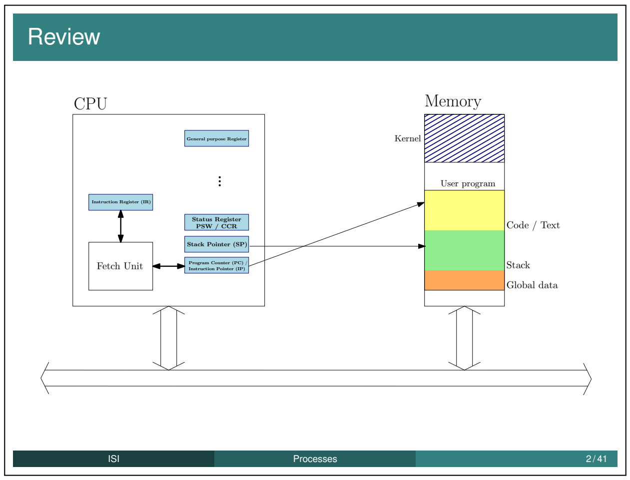
Processes

Indian Statistical Institute

1 Computer Organisation: quick review



Where is a program stored?

- The program itself (both the plain-text source (say `program.c`) and the executable file (say `a.out` generated by the compiler from the source) reside on the hard disk.
- The executable `a.out` has to be *loaded* (copied) into memory before it can be executed.
- *Address space* $\triangleq$ region of memory occupied by a running program
- Address space typically consists of the following sections (see picture above):
  - *code*: stores the sequence of machine instructions generated by the compiler from `program.c`
  - *data*: stores all global variables
  - *stack*: stores all local variables
  - *heap*: used for dynamic memory management (e.g., via `malloc()`, `free()`)

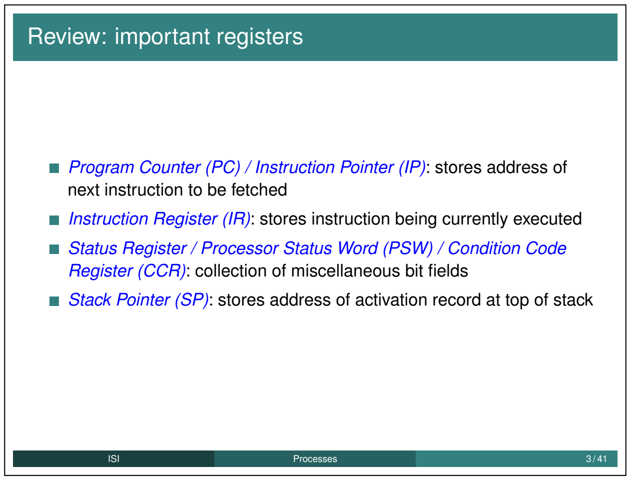
  In x86 processors, an address (memory location) consists of two parts: the first part specifies which of the the above sections the location belongs to, and the second part specifies the offset (or serial number of the location) *within* that section. More details about this addressing scheme will be discussed in the Memory Management chapter, but the notion of a 2-part address will be needed to understand some of the details given below.

Running a program

- Each instruction is a sequence of 0s and 1s.
- *Opcode* $\equiv$ part of the above sequence that specifies what operation is to be performed
  (other parts of the sequence may specify *operands*, e.g., as register numbers, memory locations)
- The circuit in the CPU continuously executes the following sequence of 3 steps in hardware:
  - FETCH: the control unit within the CPU copies an instruction from memory into the CPU.

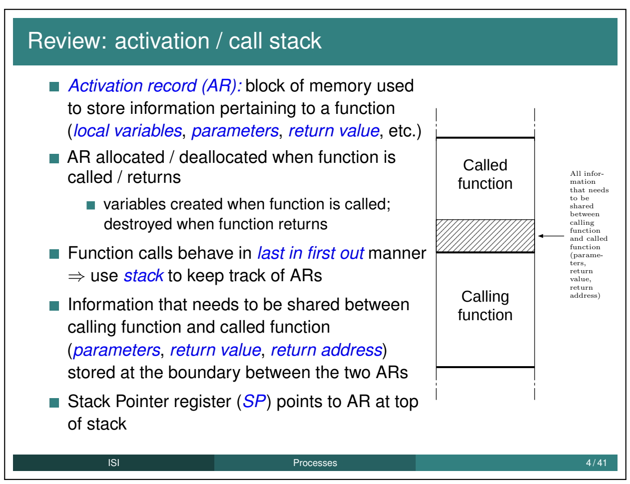
– DECODE: 'understands' what the instruction is supposed to do, and which parts of the CPU circuit need to be activated.
– EXECUTE: actually does the specified operation.

See Section 2 for a slightly more detailed version of the above FETCH-DECODE-EXECUTE cycle.



Additional references:

https://ee.usc.edu/~redekopp/cs356/slides/CS356Unit4_x86_ISA.pdf, slides 4–18
https://youtu.be/jx-w2o-Lj8g



Activation / call stack

- Activation records are also called *stack frames*.
- Additional code required to push and pop ARs is inserted by the compiler when it translates function calls / returns.

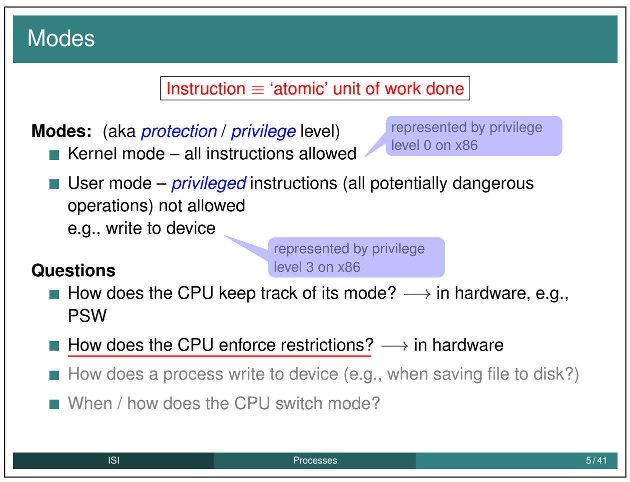
Example:

```
char *s = "Hello";
int i;
for (i = 0; i < 3; i++) {

    printf(s);

}
```

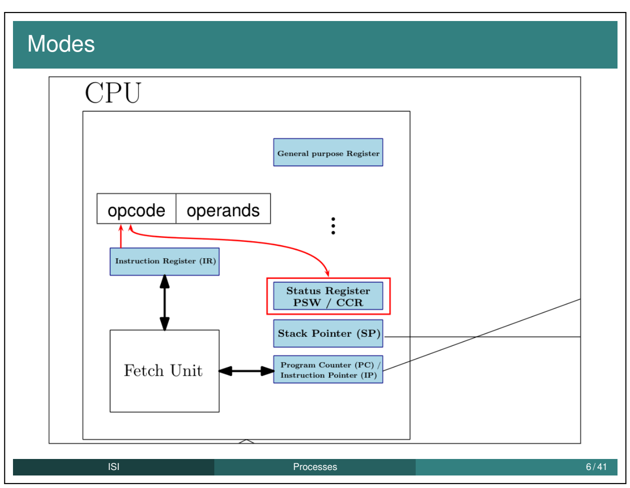
When translating this high-level code, the compiler will insert additional machine instructions (called the *calling sequence*) to allocate space on the stack above the AR for `main()` to store the AR for `printf()`. Similarly, another sequence of instructions (called the *return sequence*) is generated for handling the return from `printf()` to `main()`.

- Memory occupied by a stack frame is not necessarily deallocated / cleared after the corresponding function returns.

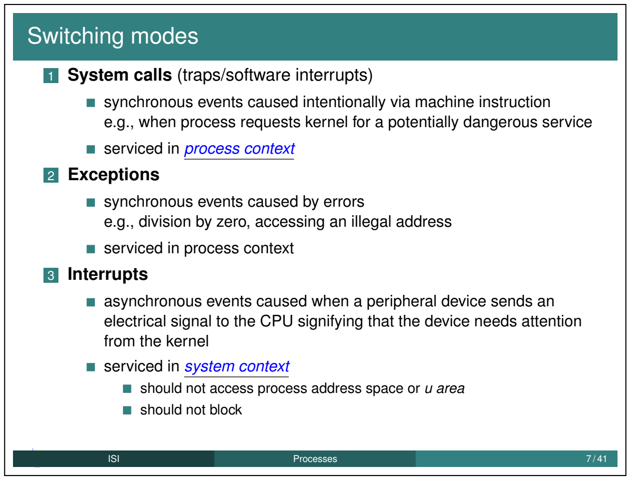
2 Processor modes



On x86 processors, the current mode / privilege level is actually stored in the `%cs` (code segment) register, but we may ignore this detail for now.



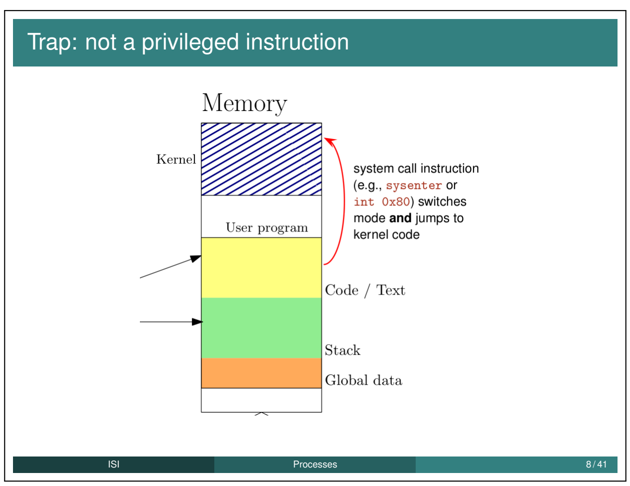
The above slide shows a very simplified example of how the processor might enforce restrictions. Suppose the processor architect adopts the convention that opcodes for privileged instructions start with 1, and the remainder start with 0. While processing the instruction, the hardware can check whether the mode bit in the PSW is "compatible" with the first bit of the opcode: if it is, execution proceeds as usual; otherwise, an exception can be raised (see below).



Synchronous vs. asynchronous events

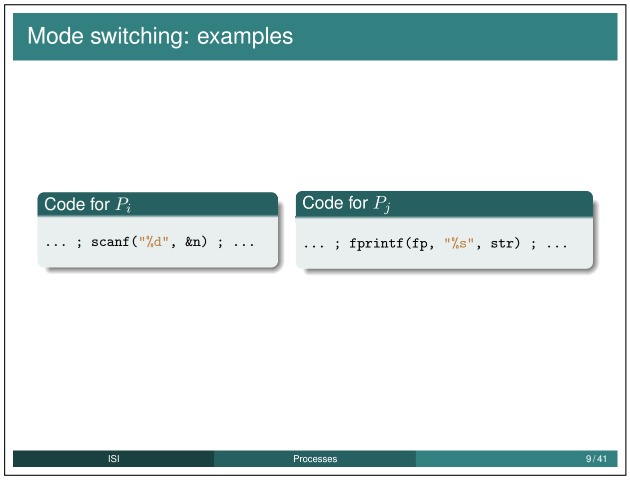
- For synchronous events, the position of the PC is predictable / well-defined. In other words, if a process P switches to kernel mode synchronously because of a system call or an exception, then it will do so at the same point in the execution of the program every time it is run (provided, of course, that the data / input values remain the same).
- Asynchronous events are caused by "external" factors, so the PC may be pointing to any arbitrary instruction of the program when the even occurs. For example, every time a user types a key, or moves the mouse, or a packet arrives over the network, an interrupt is generated. These events may not have any connection to the program that is running when the event occurs.

For analogous reasons, system calls and exceptions are said to be handled in *process context*, because the work that the kernel does is related to the process that caused the mode switch. In contrast, when an interrupt arrives, the work that the kernel does to handle the interrupt may not be related to the process running at that time, so it is said to service the interrupt in *system context*.



NOTE: The instruction that causes a mode switch cannot be privileged. If it were, then a process in user mode would not be permitted to execute the instruction, and would never be able to enter kernel mode. The mode switch instruction switches mode **and** simultaneously jumps to a location within the kernel. So, after the mode switch, kernel code is running, not the original user program. Thus, any user program can switch to kernel mode by running this unprivileged instruction, but cannot control what happens after the mode switch.

Mode switching: examples

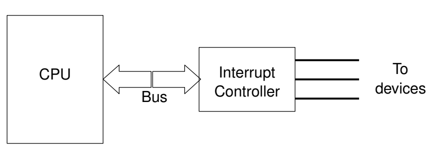


- Processes P_i and P_j call `scanf()` / `fprintf()`, which involves reading from the keyboard device / writing to a file on the hard disk. These are “potentially dangerous” operations, so the code for `scanf()` / `fprintf()` makes a system call to the kernel with a read / write request. Once the kernel gets control, its activities need not be limited to only servicing the request; it can take care of any work related to system management. Specifically, it can decide that some other process P_k needs to run, and temporarily stop P_i or P_j and switch to P_k instead.
- In contrast, consider the code below.

```
1 int i = 0;
2 while (1) i++;
3 printf("%d\n", i);
```

  A process corresponding to the above code runs an infinite loop at line 2, which does not involve any system calls or exceptions. Nevertheless, this process cannot monopolise the processor: because the timer device sends interrupts to the CPU at regular intervals, the processor is guaranteed to switch to kernel mode periodically, allowing the OS to gain control.

Updated description of FETCH-DECODE-EXECUTE cycle: During the FETCH stage, the processor first checks the incoming `INTR` line (wire) to determine whether an interrupt has arrived. If so, the control unit fetches the next instruction from a designated location in kernel code (see Section 3 for details about how the location is determined), instead of the instruction pointed to by the PC/IP register.



Usually, multiple devices are connected to an Interrupt Controller circuit (or an Advanced Programmable Interrupt Controller in modern multi-processor systems), which in turn is connected to the CPU. The Interrupt Controller receives signals from devices and conveys the interrupt to the CPU via the `INTR` line.

3 System calls, exceptions, interrupts

Interrupt Descriptor Table (IDT)

Kernel ("library") entry points for x86

- Upto 256 different entry points into the kernel
- Each entry point corresponds to a system call, exception or interrupt.
- Entry points identified by *interrupt vector* (8 bit unsigned integer)
- Different devices, error conditions, application requests, etc. generate interrupts with different vectors.
- Addresses of entry points stored in *Interrupt Descriptor Table* (IDT)
- Base address of IDT stored in `idtr` register

Kernel entry points for x86: examples

#	Interrupt/exception
0	**Divide error** (integer division by 0)
4	**Overflow** (`into` (check for overflow) instruction has been executed while the `OF` (overflow flag) of `eflags` is set)
6	**Invalid opcode**
14	**Page fault** (more about this later)
17	**Alignment check** (e.g., address of long integer not multiple of 4)
32–127	External interrupts (IRQs)
128	System calls
129–238	External interrupts (IRQs)

NOTE: To begin executing kernel code, a process must start from one of these entry points.

What happens inside the processor during a mode switch?

The CPU hardware executes roughly the same sequence of actions for system calls, exceptions and interrupts, i.e., the sequence described below may be initiated by an explicit instruction (`int`), or by hardware itself because (i) it detects an error / exception, or (ii) an interrupt has arrived via the `INTR` pin of the processor.

Hardware actions during mode switch (in x86)

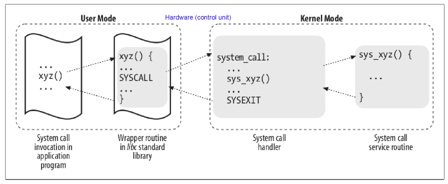
1. Processor determines the vector (number $\in \{0, \ldots, 255\}$) associated with the interrupt.
   - system calls $\equiv$ software interrupts caused by an instruction
     - operand specifies interrupt vector
   - interrupts
     - interrupt vector stored in register within the Interrupt Controller
     - CPU reads this register to determine which interrupt has arrived
2. Reads corresponding entry of IDT to get address of handler.
3. Switches from user stack to kernel stack.
   - 3.1 saves current SP register in CPU-internal registers (why?)
   - 3.2 sets up SP to point to kernel stack
   - 3.3 saves old SP (pointing to user stack) onto kernel stack
4. Pushes current PSW and IP onto new stack.
5. Loads IP with the address read in Step **??**, i.e., jumps to the appropriate kernel entry point.

User stack vs. kernel stack

The `int` instruction cannot use the user stack to save values, because the process may not have a valid stack pointer; instead, the hardware uses the stack specified in the task segment, which is set by the kernel.

An operating system can use the `iret` instruction to return from an `int` instruction. It pops the values saved during the `int` instruction from the stack, and resumes execution at the saved PC.

System calls in Linux



- System calls in user programs actually map to wrapper routines in the standard C library.
- `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”

Actions performed after `int 0x80` instruction

```
system_call: # 128th entry of IDT points here
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
```


System calls numbers

- *Dispatch table* (`sys_call_table[NR_syscalls]` array) holds addresses of service routine corresponding to each system call number

  `NR_syscalls` = 289 in the Linux 2.6.11 kernel
- Kernel looks up *dispatch vector* (system call number) in `sys_call_table` to find address of appropriate handler
- Several library functions can map into one system call

Parameter passing and return value

- System calls cross from user to kernel mode
  ⇒ neither stack can be used
  (working with two stacks at the same time is complex)
  ⇒ 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

Example convention for passing parameters / 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.

Interrupt handling

Interrupts can come anytime, when the kernel may want to finish something else it was trying to do. The kernel's goal is therefore to get the interrupt out of the way as soon as possible and defer as much processing as it can. For instance,

The activities that the kernel needs to perform in response to an interrupt are thus divided into a critical urgent part that the kernel executes right away and a deferrable part that is left for later.

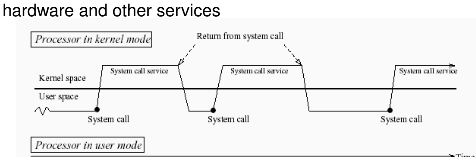
Modes: kernel vs. processes

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



4 Process context

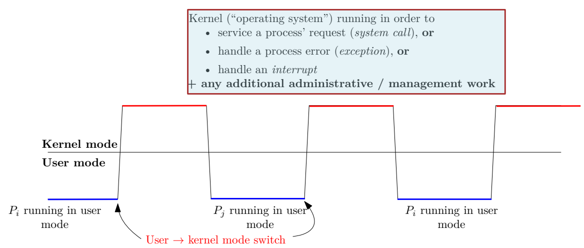
Definition

Process: an executing instance of a program

Process vs. program:

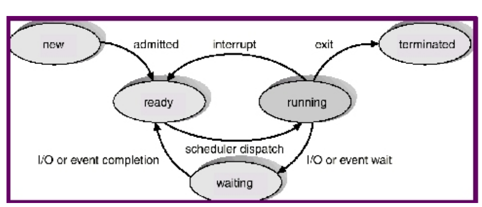
- program is static (resides in file)
- many processes may correspond to the same program (e.g. `ls`, `pine`, etc.)

Recap.



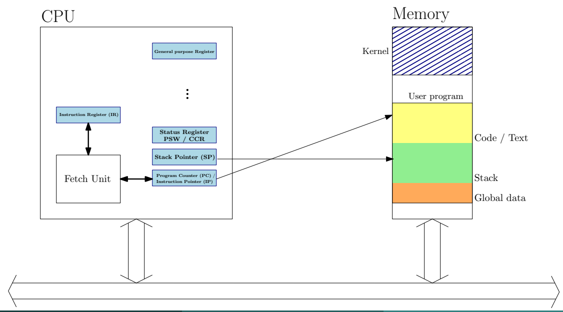
Multitasking: process states

- 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

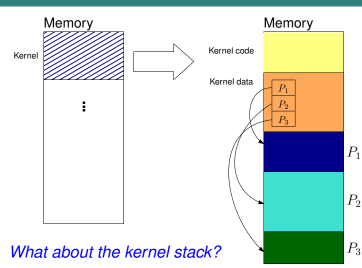
Definition: “snapshot”, i.e. complete information about a process at some point during its execution



Process context: 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

Digression: kernel memory



What about the kernel stack?

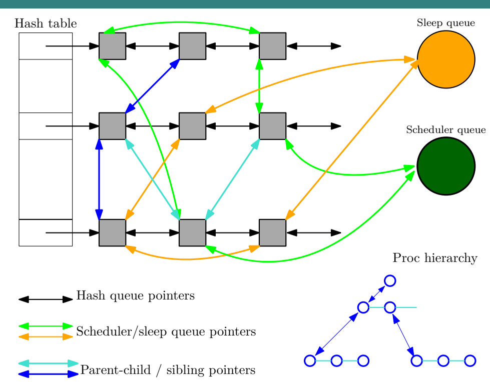
Process context: constituents (contd.)

3. **Kernel stack**
   - has to be separate for each process
   - 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 administrative information about processes
   - *proc* structure – in kernel space (always visible to kernel)
   - *u area* – in process space (visible only for running process)
6. **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)

Proc hierarchy, hash table, scheduler queues



u area

1. Pointer to *proc* structure
2. **Credentials** – Real and effective user ID (UID), group ID (GID)
3. saved register values 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

ISI Processes 28 / 41

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

⇐

ISI Processes 29 / 41

Process definition revisited

process ≡ process context ≡ set of data structures

proc structure, u area, address space (memory regions), etc.

ISI Processes 30 / 41

Digression: confusing terminology

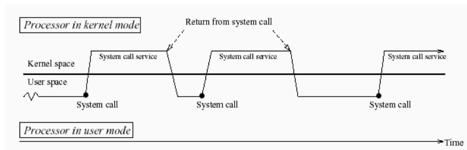
Process Control Block (PCB)

According to the textbook (Section 3.1.3): *Each process is represented in the operating system by a process control block.* This seems to suggest that the PCB corresponds to item 5 (Control information) in the list of constituents that make up a process context. In other words, it corresponds to a combination of the proc structure and the u area. This is consistent with the description in the infobox titled "PROCESS REPRESENTATION IN LINUX" under Section 3.1.4, which reads: *The process control block in the Linux operating system is represented by the C structure* `task_struct` ...

According to Vahalia (Sections 2.3.2, 2.3.4, 5.1), the PCB is a special part of the u area that saves the hardware context (i.e., the values of all registers) of a process at the time of a process switch.

5 Process switching

Process switching



- 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

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.

Hyperthreading

- Modern processors have *multiple sets* of registers
- One set of registers in active use at a time
- Process switching may involve only switching the active set (no memory operations needed)
- Conventional save-and-restore memory operations needed when no. of active processes exceeds no. of register sets,

ISI Processes 34 / 41

How much time does a context switch take? Typical times range from a few milliseconds to a few microseconds.

Process switching in Linux

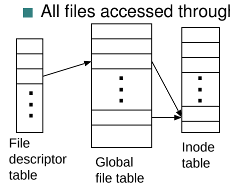
`kernel/sched/core.c`

ISI Processes 35 / 41

6 Process related system calls

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)

ISI Processes 36 / 41

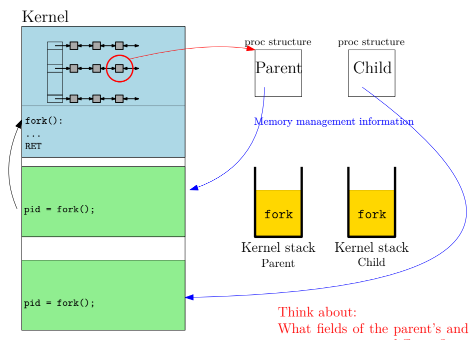
Process creation

Syntax: `pid = fork();`
`pid` – PID of child process (parent)
`pid` – 0 (child)

Usage

```
pid = fork();
if (pid < 0) exit(1);
if (pid == 0) {
   /* child process executes this code */
}
else {
   /* parent process executes this code */
}
```


Process creation



Think about:
What fields of the parent's and child's proc structure are different?

Process creation

Algorithm:

I. **Preliminary checks**
   1. Check for available memory.
   2. Check that user is not running too many processes.

II. **Create + initialise a clone of the parent**
   1. Allocate new *proc* structure, assign new PID.
   2. 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
   3. Clear accounting information, timers, pending signals.
   4. Connect new *proc* on relevant linked lists.

PIDs start from 1 and increase by 1 until wraparound at maximum value

Process creation

II. **Create + initialise a clone of the parent** (CONTD.)

5. Allocate memory and create copy of parent context (*u area*, regions, page tables)
   - shared regions are not copied, only ref. count is incremented
   - *u area* contains user FD table
   - child inherits access rights to open files
   - child shares global file table entries with parent
   - changes in file offset caused by read/write in the parent are visible to child and vice versa
6. Copy parent's kernel-level context (registers + kernel stack).

III. **Book-keeping**

1. Increment reference count of inode of current directory.
2. Increment global file table reference count associated with each open file of parent process.

Wh

Process creation

IV. **Distinguish between parent and child:**
Parent: return PID to user
Child: "saved" context is restored, returns 0 to user

Process termination

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

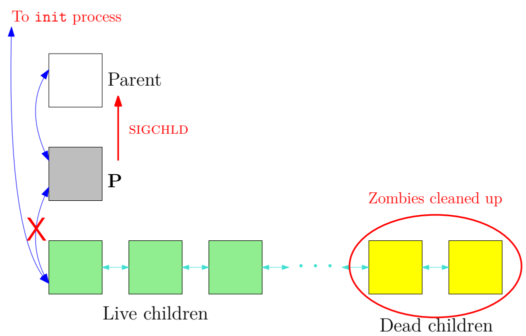
Algorithm:

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.

Process termination



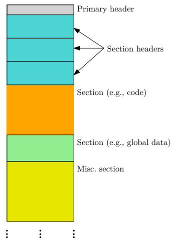
Invoking a program

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

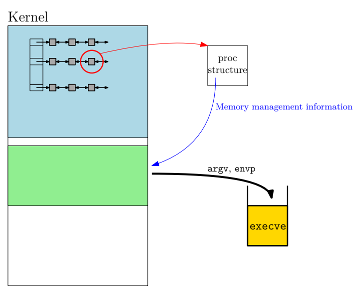


2. **Handle old address space.**
   1. Copy parameters from old address space to kernel space.
      (old address space will be freed ⇒ params have to be saved on:
      kernel stack + additional storage (if needed))
   2. Free memory occupied by the process.

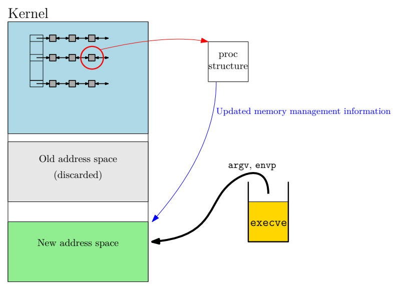
Invoking a program

3. **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).

Invoking a program



Invoking a program



Summary

- Relation between processes and the kernel (“operating system”)
- User mode vs. kernel mode, mode switches
- 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 ≡ its *context* (complete information about the process at any point during its execution)
- `fork`, `exit`, *exec* system calls