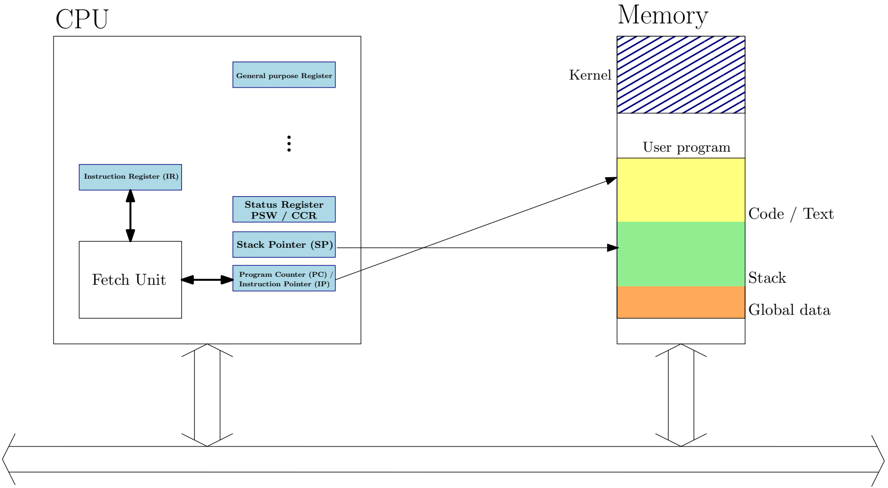
Processes

Indian Statistical Institute

https://www.isical.ac.in/~mandar/courses.html#os

Review



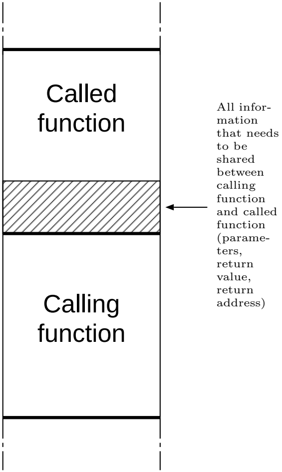
CPU
General purpose Register
Instruction Register (IR)
Status Register
PSW / CCR
Stack Pointer (SP)
Fetch Unit
Program Counter (PC) /
Instruction Pointer (IP)
Memory
Kernel
User program
Code / Text
Stack
Global data

Review: important registers

- *Program Counter (PC) / Instruction Pointer (IP)*: stores address of next instruction to be fetched
- *Instruction Register (IR)*: stores instruction being currently executed
- *Status Register / Processor Status Word (PSW) / Condition Code Register (CCR)*: collection of miscellaneous bit fields
- *Stack Pointer (SP)*: stores address of activation record at top of stack

Review: activation / call stack

- *Activation record (AR):* block of memory used to store information pertaining to a function (*local variables*, *parameters*, *return value*, etc.)
- AR allocated / deallocated when function is called / returns
  - variables created when function is called; destroyed when function returns
- Function calls behave in *last in first out* manner ⇒ use *stack* to keep track of ARs
- Information that needs to be shared between calling function and called function (*parameters*, *return value*, *return address*) stored at the boundary between the two ARs
- Stack Pointer register (*SP*) points to AR at top of stack



Modes

Instruction $\equiv$ 'atomic' unit of work done

Modes: (aka *protection* / *privilege* level)

- Kernel mode – all instructions allowed

  represented by privilege level 0 on x86

- User mode – *privileged* instructions (all potentially dangerous operations) not allowed
  e.g., write to device

  represented by privilege level 3 on x86

Modes

Instruction $\equiv$ 'atomic' unit of work done

Modes: (aka *protection* / *privilege* level)

- Kernel mode – all instructions allowed

represented by privilege level 0 on x86

- User mode – *privileged* instructions (all potentially dangerous operations) not allowed
  e.g., write to device

represented by privilege level 3 on x86

Questions

- How does the CPU keep track of its mode?

Modes

$\boxed{\text{Instruction} \equiv \text{'atomic' unit of work done}}$

Modes: (aka *protection* / *privilege* level)

- Kernel mode – all instructions allowed

represented by privilege level 0 on x86

- User mode – *privileged* instructions (all potentially dangerous operations) not allowed
  e.g., write to device

represented by privilege level 3 on x86

Questions

- How does the CPU keep track of its mode? $\longrightarrow$ in hardware, e.g., PSW

Modes

Instruction $\equiv$ 'atomic' unit of work done

Modes: (aka *protection* / *privilege* level)

- Kernel mode – all instructions allowed

represented by privilege level 0 on x86

- User mode – *privileged* instructions (all potentially dangerous operations) not allowed
  e.g., write to device

represented by privilege level 3 on x86

Questions

- How does the CPU keep track of its mode? $\longrightarrow$ in hardware, e.g., PSW
- How does the CPU enforce restrictions?

Modes

$\boxed{\text{Instruction} \equiv \text{`atomic' unit of work done}}$

Modes: (aka *protection* / *privilege* level)

- Kernel mode – all instructions allowed

represented by privilege level 0 on x86

- User mode – *privileged* instructions (all potentially dangerous operations) not allowed
  e.g., write to device

represented by privilege level 3 on x86

Questions

- How does the CPU keep track of its mode? $\longrightarrow$ in hardware, e.g., PSW
- How does the CPU enforce restrictions? $\longrightarrow$ in hardware

Modes

Instruction $\equiv$ 'atomic' unit of work done

Modes: (aka *protection* / *privilege* level)

- Kernel mode – all instructions allowed

represented by privilege level 0 on x86

- User mode – *privileged* instructions (all potentially dangerous operations) not allowed
  e.g., write to device

represented by privilege level 3 on x86

Questions

- How does the CPU keep track of its mode? $\longrightarrow$ in hardware, e.g., PSW
- How does the CPU enforce restrictions? $\longrightarrow$ in hardware
- How does a process write to device (e.g., when saving file to disk?)
- When / how does the CPU switch mode?

Modes

Instruction $\equiv$ 'atomic' unit of work done

Modes: (aka *protection* / *privilege* level)

- Kernel mode – all instructions allowed

represented by privilege level 0 on x86

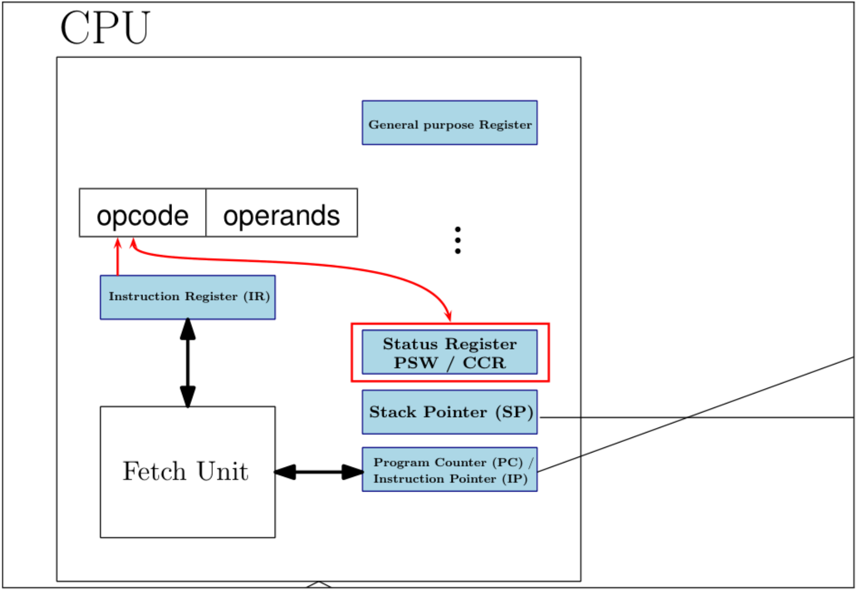
- User mode – *privileged* instructions (all potentially dangerous operations) not allowed
  e.g., write to device

represented by privilege level 3 on x86

Questions

- How does the CPU keep track of its mode? $\longrightarrow$ in hardware, e.g., PSW
- How does the CPU enforce restrictions? $\longrightarrow$ in hardware
- How does a process write to device (e.g., when saving file to disk?)
- When / how does the CPU switch mode?

Modes


CPU
General purpose Register
opcode
operands
Instruction Register (IR)
Status Register
PSW / CCR
Stack Pointer (SP)
Fetch Unit
Program Counter (PC) /
Instruction Pointer (IP)

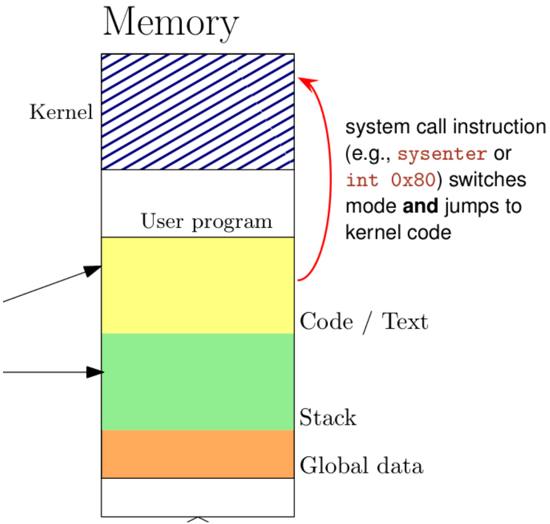
Switching modes

1. **System calls** (traps/software interrupts)
   - synchronous events caused intentionally via machine instruction
     e.g., when process requests kernel for a potentially dangerous service
   - 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 when a peripheral device sends an electrical signal to the CPU signifying that the device needs attention from the kernel
   - serviced in *system context*
     - should not access process address space or *u area*
     - should not block

Switching modes

1. **System calls** (traps/software interrupts)
   - synchronous events caused intentionally via machine instruction
     e.g., when process requests kernel for a potentially dangerous service
   - 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 when a peripheral device sends an electrical signal to the CPU signifying that the device needs attention from the kernel
   - serviced in *system context*
     - should not access process address space or *u area*
     - should not block

Trap: not a privileged instruction



Mode switching: examples

Code for P_i

```
... ; scanf("%d", &n) ; ...
```

Code for P_j

```
... ; fprintf(fp, "%s", str) ; ...
```

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

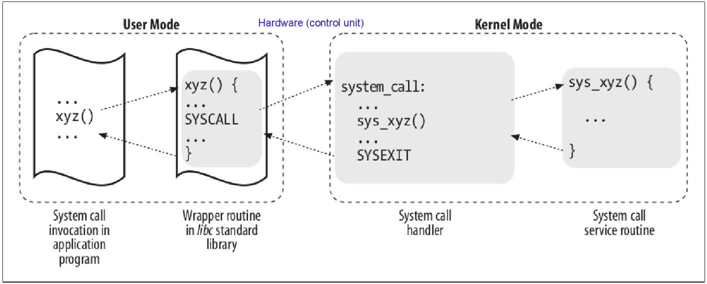
Kernel entry points for x86: examples

#	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)

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.

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
  $\Rightarrow$ 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

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.

Modes: kernel vs. processes

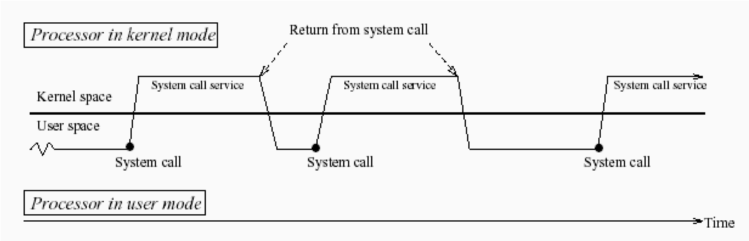
Kernel:

- 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



Definition

Process: an executing instance of a program

Process vs. program:

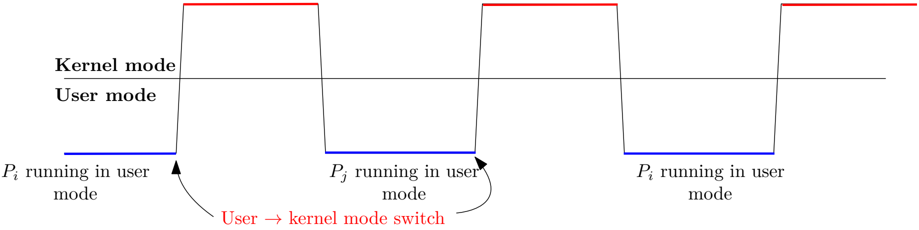
- program is static (resides in file)
- many processes may correspond to the same program (e.g. `ls`, `pine`, etc.)

Recap.

Kernel (“operating system”) running in order to

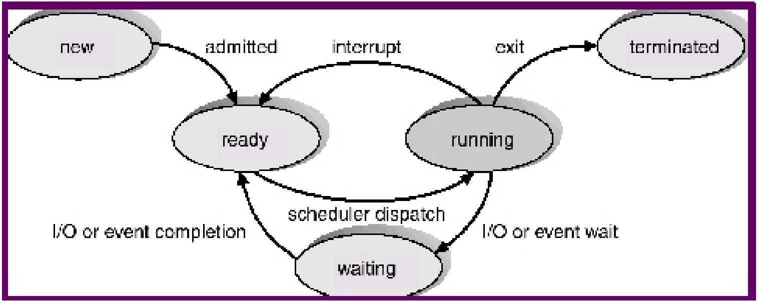
- service a process’ request (*system call*), **or**
- handle a process error (*exception*), **or**
- handle an *interrupt*

+ **any additional administrative / management work**



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



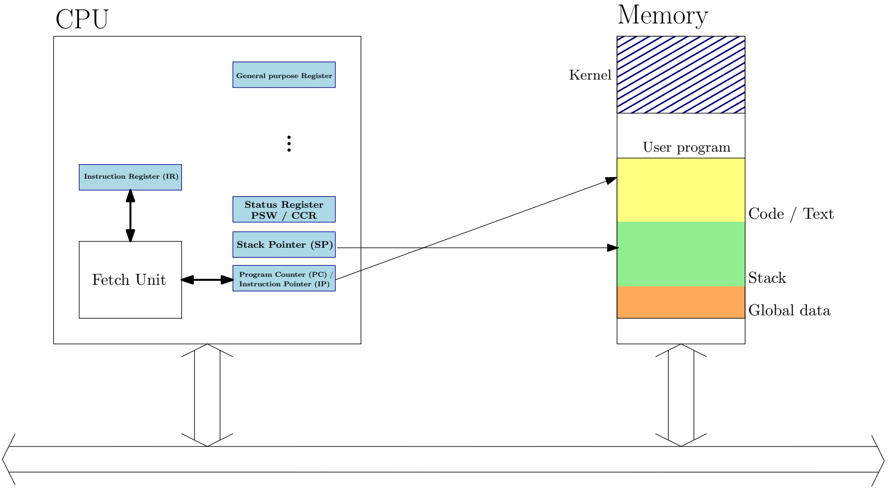
▶

Process context

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

Process context

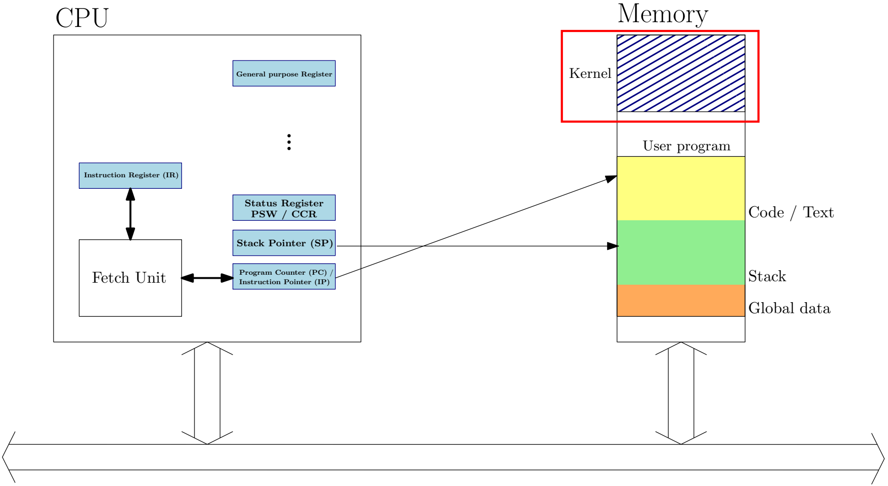
Definition: "snapshot", i.e. complete information about a process at some point during its execution



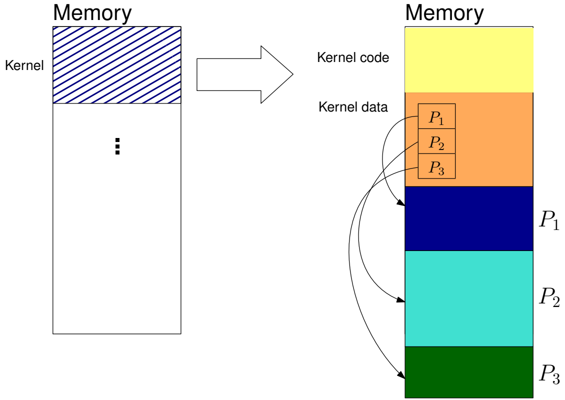
Process context: constituents

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

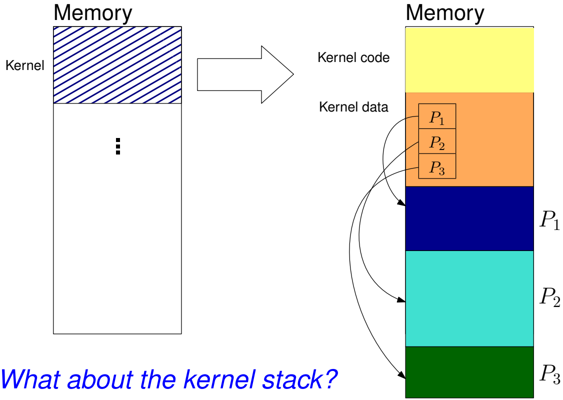
Digression: kernel memory



Digression: kernel memory



Digression: kernel memory



What about the kernel stack?

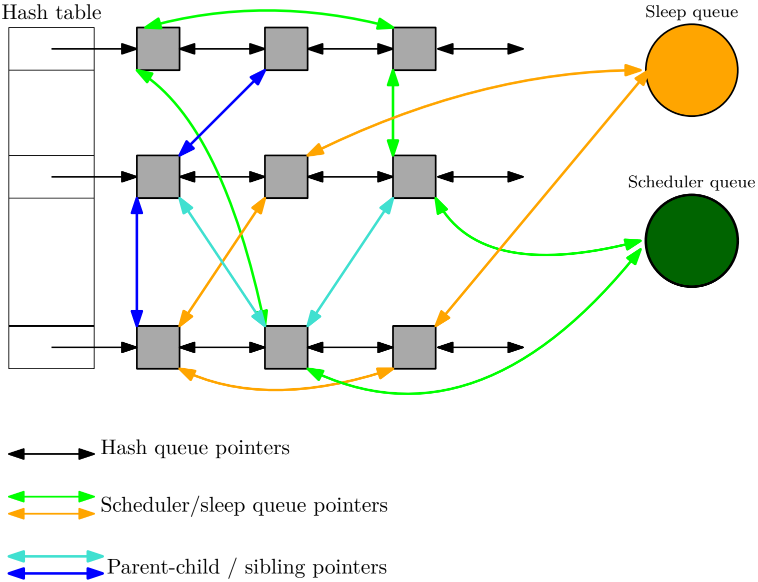
Process context: constituents (contd.)

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

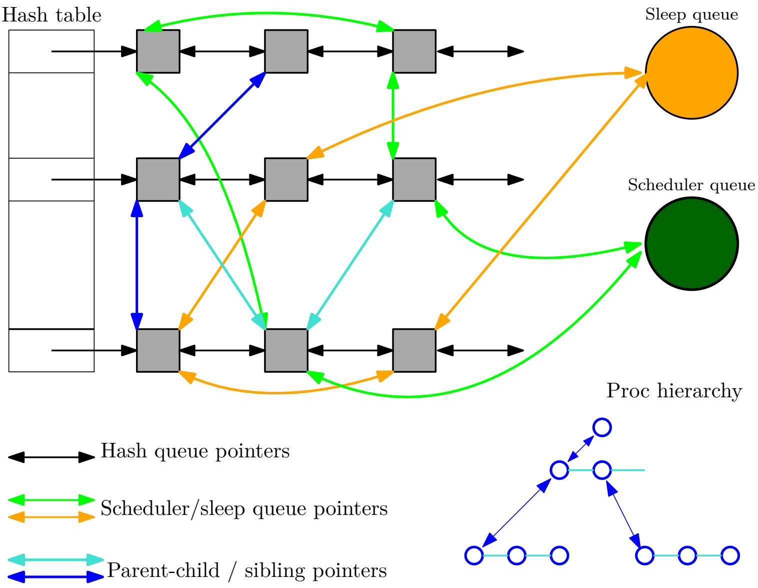
proc structure

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



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

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

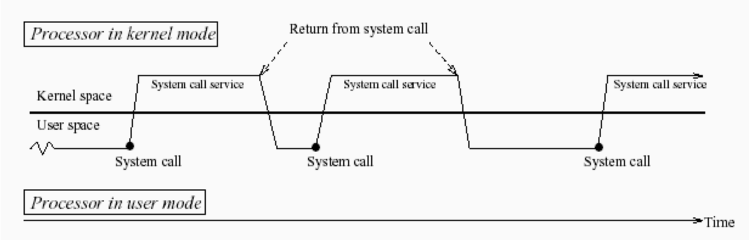
$\Leftarrow$

Process definition revisited

process $\equiv$ process context $\equiv$ set of data structures

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

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

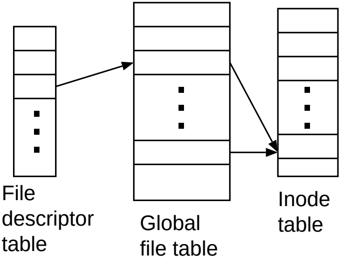
- Modern processors have *multiple sets* of registers
- One set of registers in active use at a time
- Process switching may involve only switching the active set (no memory operations needed)
- Conventional save-and-restore memory operations needed when no. of active processes exceeds no. of register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

Process switching in Linux

`kernel/sched/core.c`

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

Syntax: `pid = fork();`

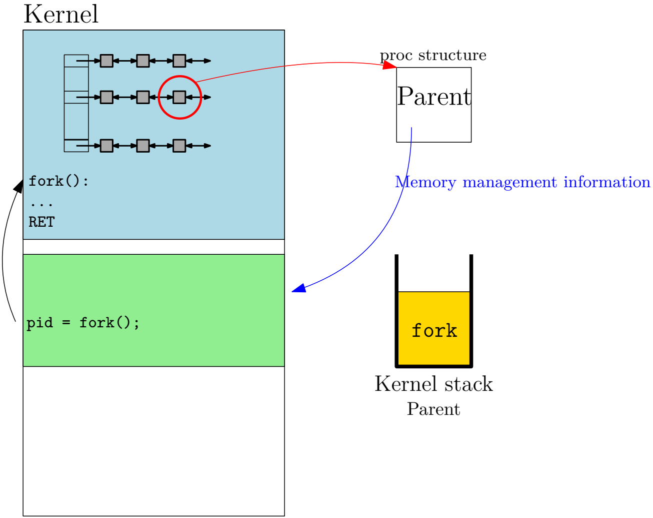
`pid` – PID of child process (parent)

`pid` – 0 (child)

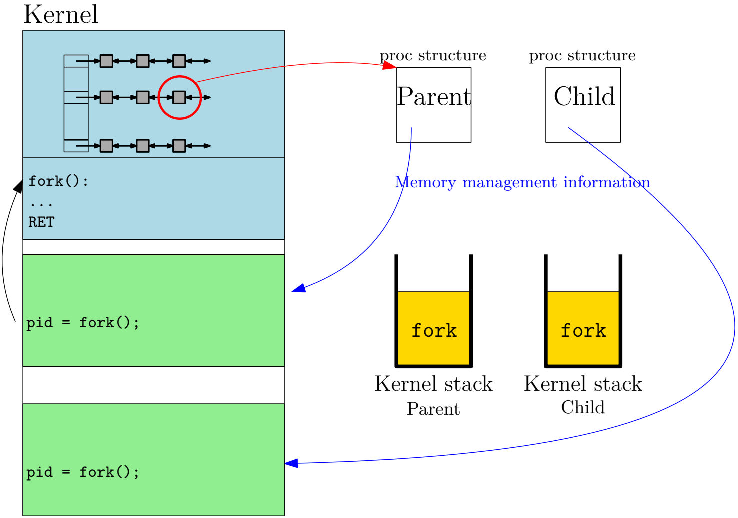
Usage

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

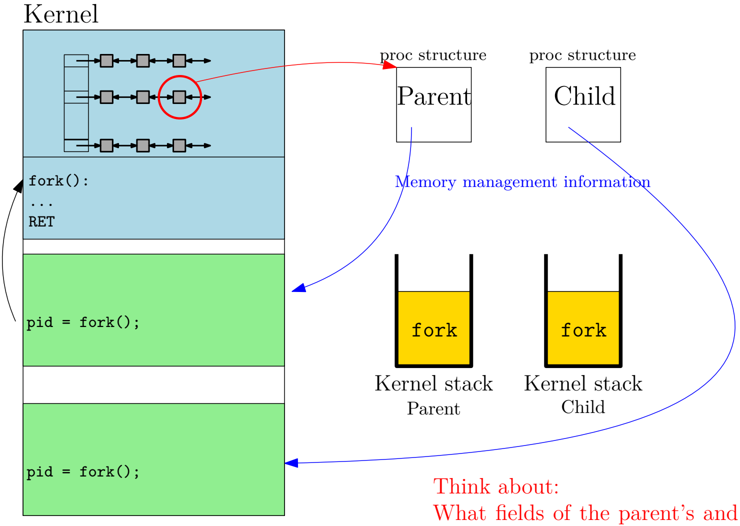
Process creation



Process creation



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 *proc*s
      - 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).

Wh

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.

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

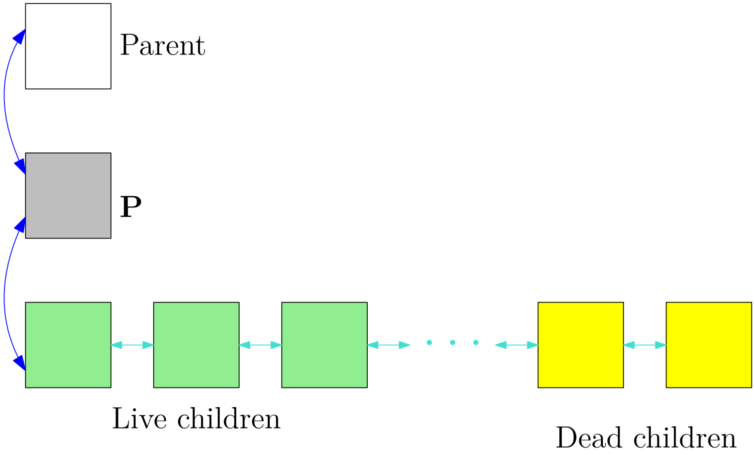
Algorithm:

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.)

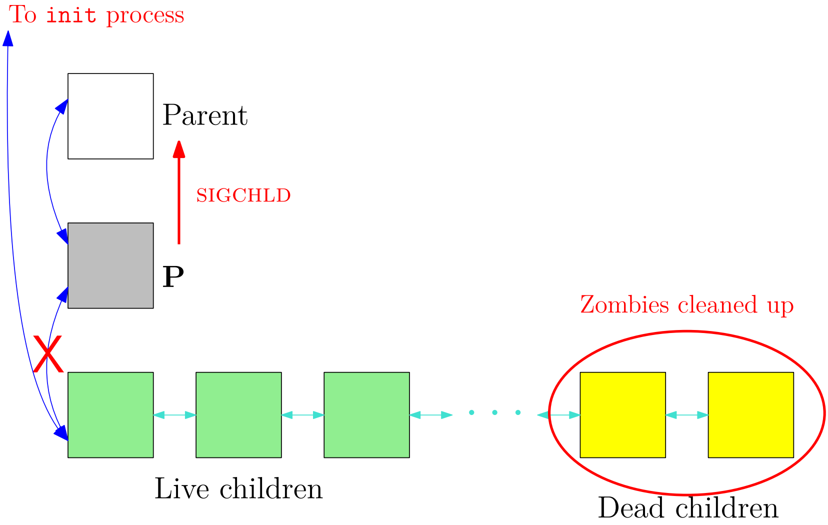
Process termination

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



Process termination



Invoking a program

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

`filename` – name of executable file

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

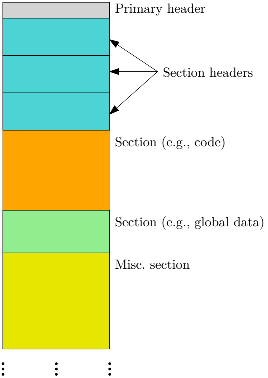
`envp` – environment of program (`char **`)

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

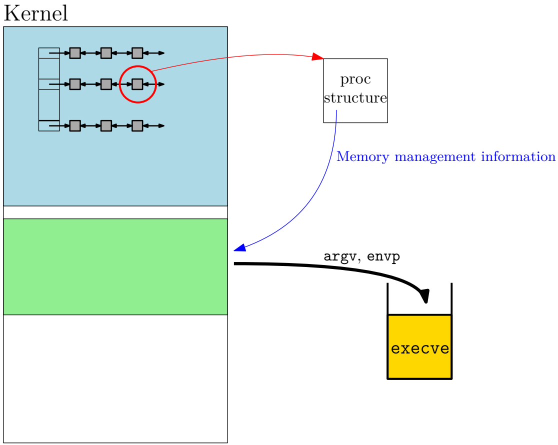


2. **Handle old address space.**
   1. Copy parameters from old address space to kernel space.
      (old address space will be freed $\Rightarrow$ params have to be saved on:
      kernel stack + additional storage (if needed))
   2. Free memory occupied by the process.

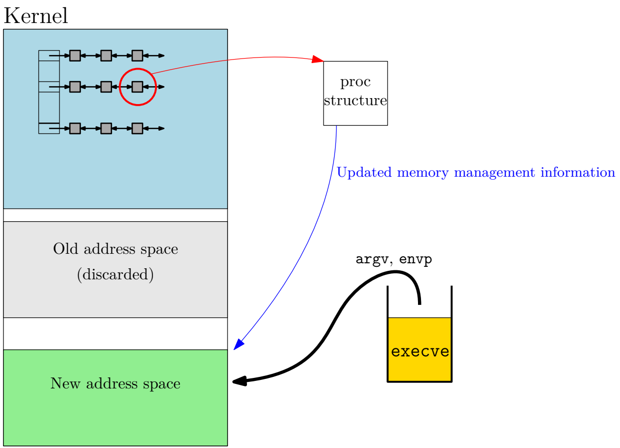
Invoking a program

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 $\equiv$ its *context* (complete information about the process at any point during its execution)
- `fork`, `exit`, *exec* system calls