System Architecture

4 Activities

Process, Task, Thread

November 3 2008

Winter Term 2008/09

Gerd Liefländer

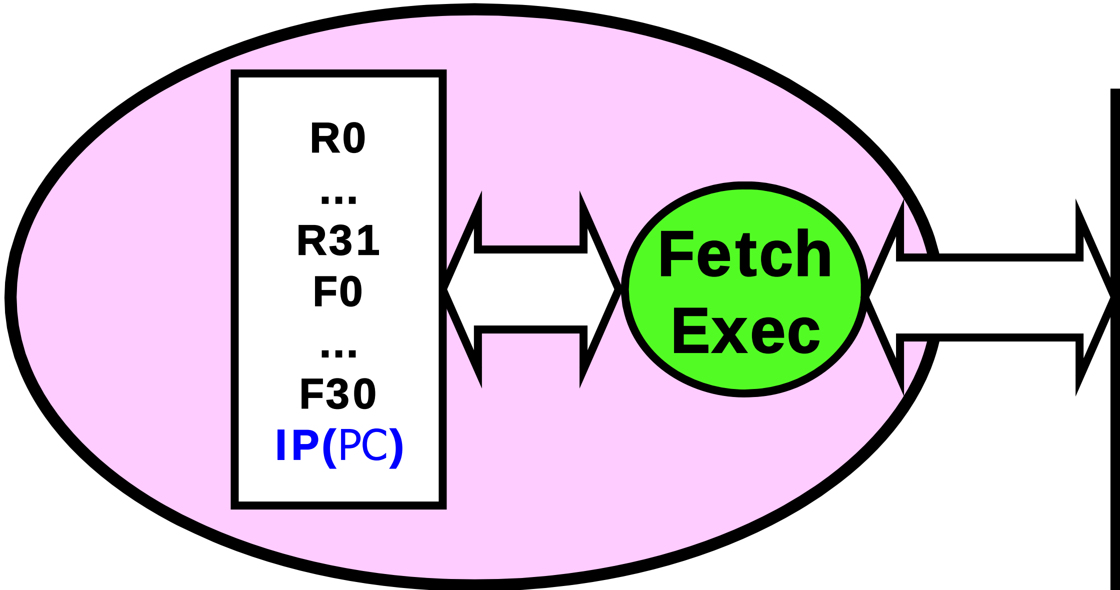
Agenda

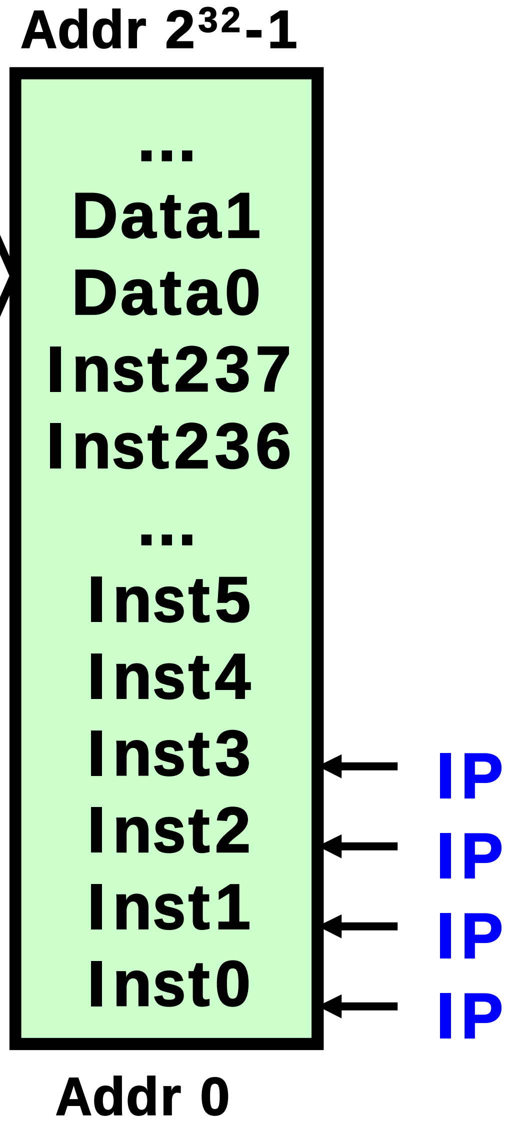
- Review
- Motivation & Introduction
- Basic Terms
- Process Model
- Task Model
- Thread Model

Nice paper: "Microsoft schrumpft Windows-Kernel"

http://www.golem.de/0710/55519.html

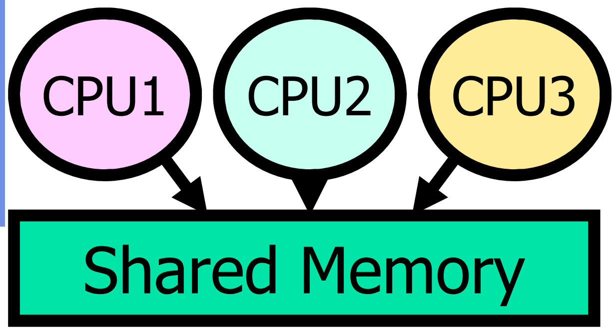
What happens during Execution?

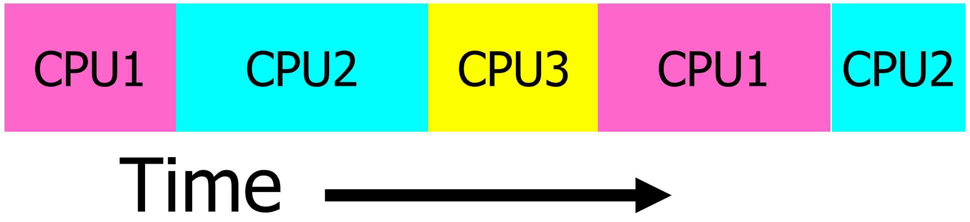




- Execution sequence:
  - Fetch Instruction at **IP**
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - **IP** = Next Instruction(**IP**)
  - Repeat

Illusion of an SMP on a Uniprocessor?





- *How to provide the illusion of multiple processors?*
  - Multiplex the CPU in time, i.e. virtualize the CPU
- Each virtual CPU needs a structure to hold:
  - Instruction Pointer (**IP**), Stack Pointer (**SP**)
  - **r>1** Registers (Integer, Floating point, others…?)
- *How to switch from one virtual CPU to the next?*
  - Save IP(PC), SP, and **r**' registers in *current context block*
  - Load IP(PC), SP, and **r'** registers from *new context block*
- *What is triggering a virtual CPU switch?*

Motivation & Introduction

Why Activities?

- HW offers concurrent execution of programs, e.g.
  - CPU(s) & I/O-devices can run in parallel
- Suppose a chess program can be parallelized, a>1 activities search in different chess data bases, another activity calculates the next moves etc. ⇒

  Better response time of the chess program

- OS has to offer appropriate concepts to enable concurrent activities:
  - Process
  - Task
  - Thread

2 Main *Abstractions* of Systems

1. How to provide „information processing", i.e. "when" to execute "what" code?

   ⇒ thread (process*)

2. How to provide „protected depositories", i.e. "where" to store "what" information entity?

   ⇒ address space (**AS**)

*<u>Note:</u> "Process" ← "lat. procedere" = "voranschreiten"
Notion "thread" ~ "Faden" abwickeln

Design Parameters: Activity

- Number of activities
  - Static systems, i.e. at run-time no new activities
  - Dynamic
- Grade of interactivity
  - Foreground, i.e. many interactions between user and activity
  - Background activity, e.g.
    - Daemons (Unix/Linux background services)
    - Applications controlled by background shell
- Urgency
  - Real time
    - Hard real time
    - Soft real time
  - Interactive
  - Batch processing

Design Parameters: Address Spaces

- Number and placement of regions
  - 1 versus n>1 regions
  - Contiguous address space
  - Non-contiguous address space
  - With(out) boundary checks
- Types of regions
  - Stack
  - Heap
  - Code
  - …
- Duration of data entity
  - Temporary
  - Persistent
- …

Basic Terms

Process

Program

Process

- Process ~ abstraction for a single sequential activity within a protected environment
  - It represents the "execution" of
    - an application program
    - a system program (outside the kernel)
  - It consists of an AS, context, state, and resources
  - Process* has only one single thread of execution
- The system entity consisting of multiple activities within one AS is a task*

*Note: This terminology is KIT specific

Program versus Process

- Same program can be executed concurrently by multiple processes, e.g. the gcc program
- Program is something static
  - It has i>1 instructions
  - These i instructions are placed somewhere in RAM(or disk)
- Process is something dynamic (sometimes a process has to wait for an event, i.e. it does no real progress)
- An executable program (e.g. file **`xyz.exe`**) has to be loaded[1] before becoming a process
- In order to control a process the OS needs a process descriptor, i.e. a process control block (**PCB**)

[1] ∃ different ways of loading a program

Process

- An executing instance of an executable program
  - ∃ only one executable **file** **gcc**
  - During a C course, simultaneously multiple **gcc**-processes can be active on one computer (e.g. at your computer center)
- Each process is separated from another executing **gcc** process
  - If one **gcc**-process **fails** it should not bother another concurrent **gcc**-process or any other concurrent application process
- Processes can start (launch) other processes

gcc Example

- Via a command you can launch a gcc process
- It first launches cpp, cc1, as
- Finally it launches ld
- Each instance is a process, and each of the above programs actually exists separately

Process Model

Process State

Process Control Block

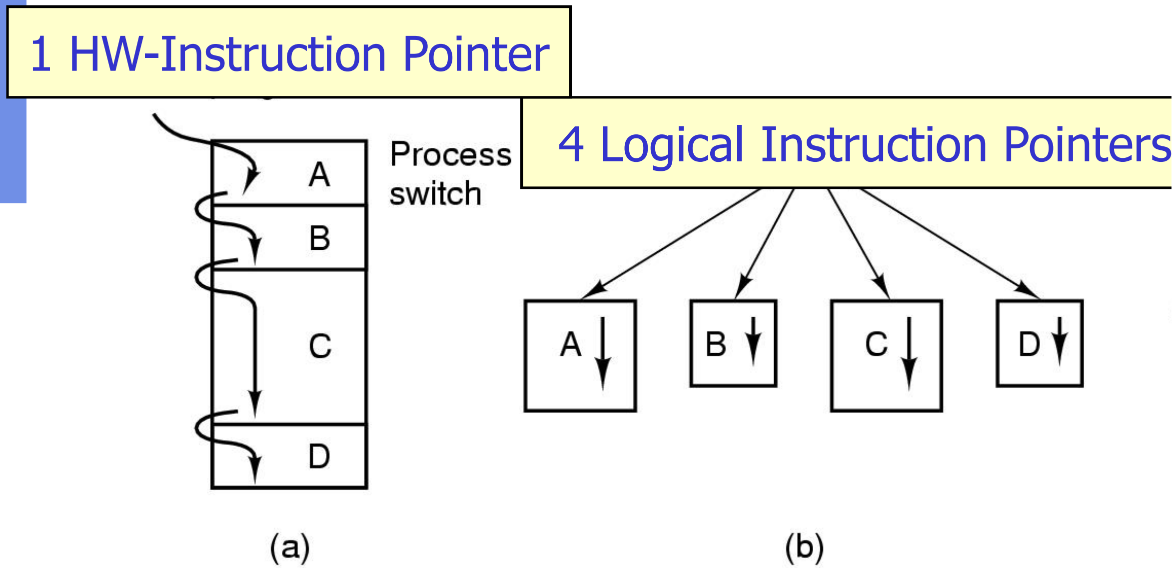
Abstract Process Switch

"Processing" Model

according to weight classes of boxers

- Heavyweight processing (e.g. Unix process)
  - Activity instance and address space form a system unit
  - Each "process switch" involves 2 AS switches:
    $AR_x \rightarrow OS_{kernel} \rightarrow AR_y$
- Lightweight processing (Kernel Level Threads, KLTs))
  - Activity instances and address space are decoupled
  - A "KLT switch" can involve 1.5 or 2 AS switches:
    $AR_x \rightarrow OS_{kernel} \rightarrow AR_{y \text{ or } x}$
- Featherweight processing (Pure User Level Threads, PULTs)
  - Activity instances and AS form a single system unit
  - A "PULT switch" at user level involves no AS switch, i.e.
    $AR_x \rightarrow Ar_x$
- Whenever a thread switch is done without the kernel, then it must be a switch between two PULTs of the same AS

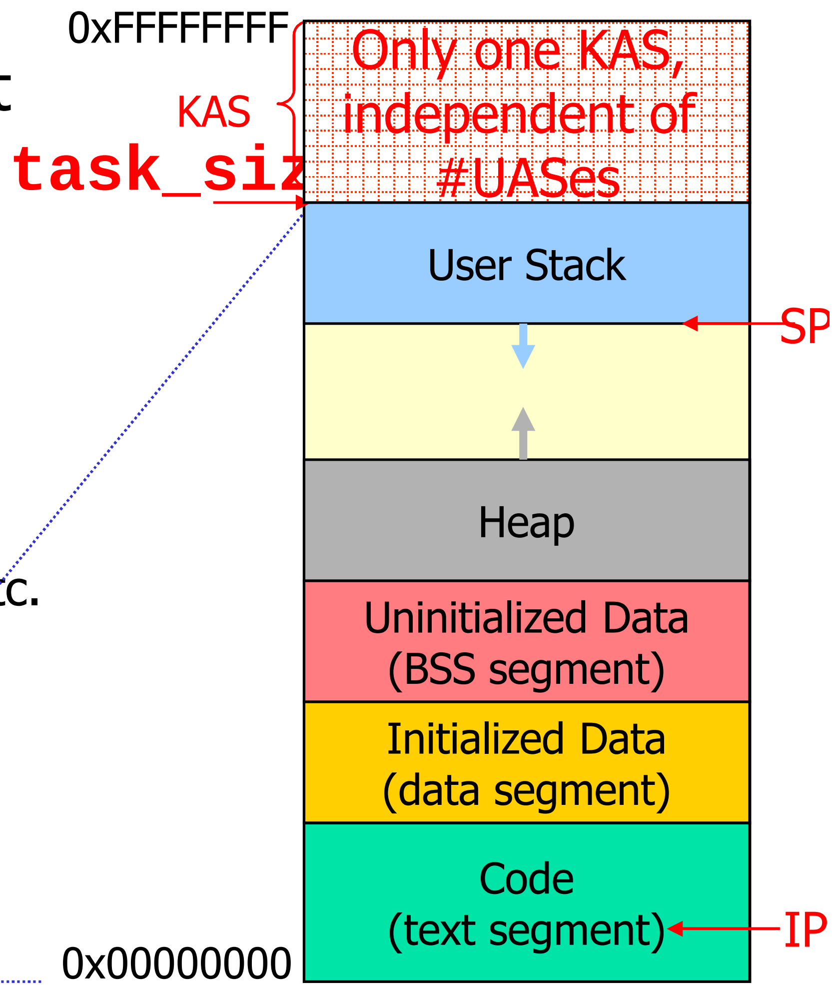
Process Model



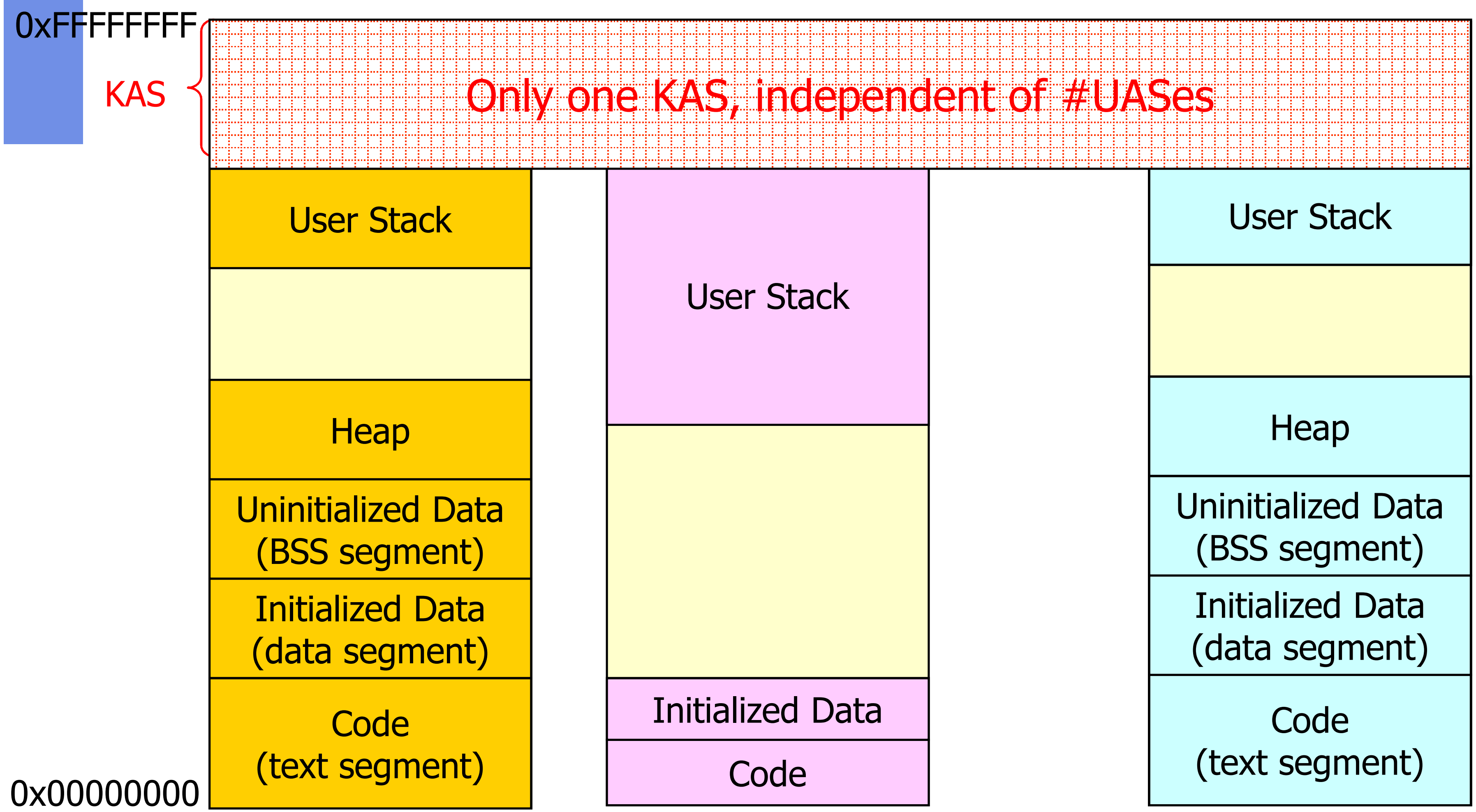
- Multiprogramming of four applications: each application is implemented as a process, i.e. in an isolated AS
- Conceptually: ∃ four independent, sequential processes
- Only one process can run at any instant of time on a conventional single-processor system

Example: AS of a Linux Process

- Logical address regions, that a process can access:
  - Code
  - Data
    - static variables
    - heap
  - User stack
    - Local variables, parameters etc. (to be installed for each call)



Multiple User Address Spaces



Process Creation

Principal events causing a creation of a process:

1. System initialization (e.g. **`sysinit`**)
2. System call by another process (e.g. **`fork`**)
3. User creates a new process via a
   1. Command (e.g. **`sh`** forks and then loads a new program image via system call **`exec(ve)`**)
   2. Click on icon representing an executable
4. Initiation of a batch job

Process Termination

Conditions that can terminate a process:

1. Normal exit (voluntary)
2. Error exit (voluntary)
   - Programmer has provided an exception handler
3. Fatal error (involuntarry)
   - System handles exception
4. Aborted by another process (involuntary)
   - *Scenarios leading to a process abortion?*

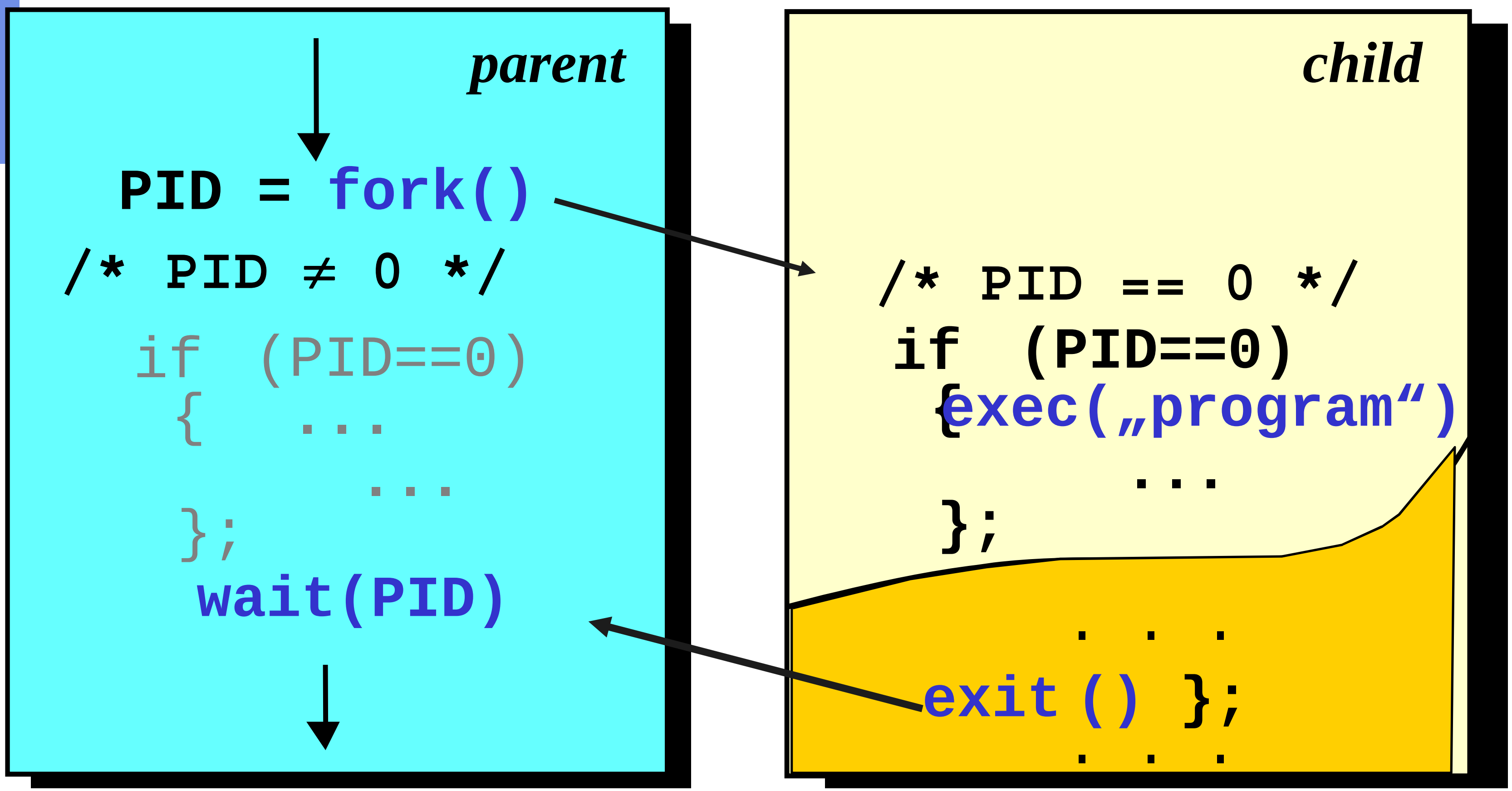
Process Hierarchy

- Parent process creates a child process, a child process can create further children
  - From the perspective of the parent these new kids are grand-children
  - …
- Forms a rooted-tree-like process hierarchy
  - UNIX calls it a "process group"
- Windows: no concept of a process hierarchy
  - All processes are created at the same level

What else can describe a Process?

- Information about process hierarchy
  - *Who has launched the process*
  - *What processes have been launched by it*
- Information about resources
  - *Where does it store its data*
  - *What other resources does it use*
- Various kinds of mappings
  - *What address regions belong to which process*

Creating a Unix Process



Hint: ∃ good introduction how to create Unix, Linux and XP processes
http://www-106.ibm.com/developerworks/linux/library/l-rt7/?Open&t=grl,l=252,p=mgth

Fork Example

```
int value = 5;                  /* a global variable */
int main() {
 pid_t pid;
 value = 7;                     /* parent */
 pid = fork();
 if (pid ==0) {                 /* child */
    value = 15;
 }
 else {                         /* parent */
    wait(NULL) /* wait for child to terminate */
    printf("PARENT: value = %d\n", value);
 }
}
```

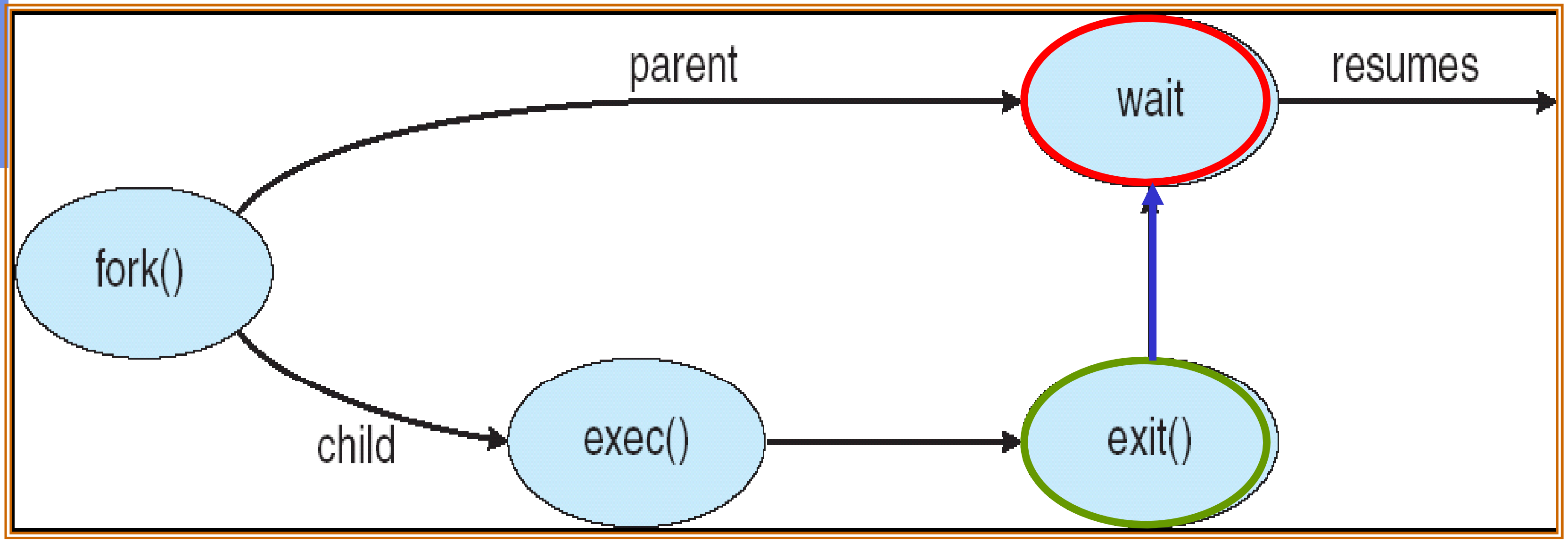
Exec versus Fork

- *So how do we start a new program, instead of just forking the old program?*
  - The systemcall **`exec()`**
  - **`int exec(char *prog, char ** argv)`**
- **`exec()`**
  - Stops the current process
  - Loads the executable program **`prog`** into AS of the caller
  - Initializes HW context, args for the new program
- Note: **`exec()`** does not create a new process

Example: Unix SHELL

```
int main(int argc, char **argv) {
 while (1){
  char *cmd =  get_next_command();
  int child_pid = fork();
  if (child pid ==0) {
     exec(cmd);
     panic("exec failed!");
  } else {
  waitpid(child_pid);
  }
 }
}
```

Process Creation in Unix/Linux



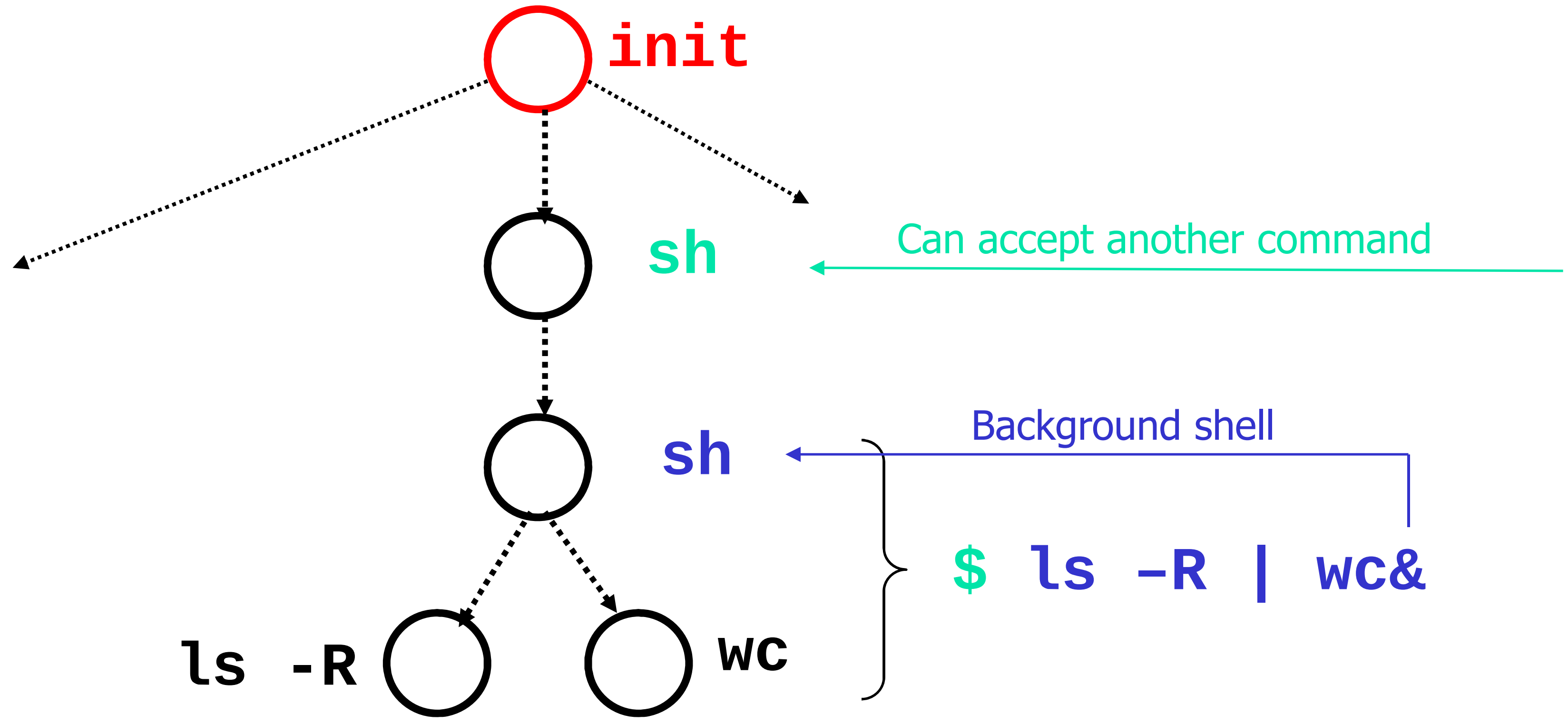
Potential problems that have to be solved in a robust OS

1. *What happens when parent dies before child exits?*
2. *What happens when parent does not wait?*
3. *What happens when child does not exit?*

Unix Processes

- Process executes last statement and asks OS to finish it via system call `exit()`
  - Output data from child to parent process waiting for the result via `wait()`
  - Resources of child can be released if no longer used otherwise
- Parent or OS can terminate execution of child process (`abort`) in case of
  - Child has exceeded allocated resources
  - Job assigned to the child is no longer required
  - Child misbehaves (looping forever)

Unix Process Hierarchy

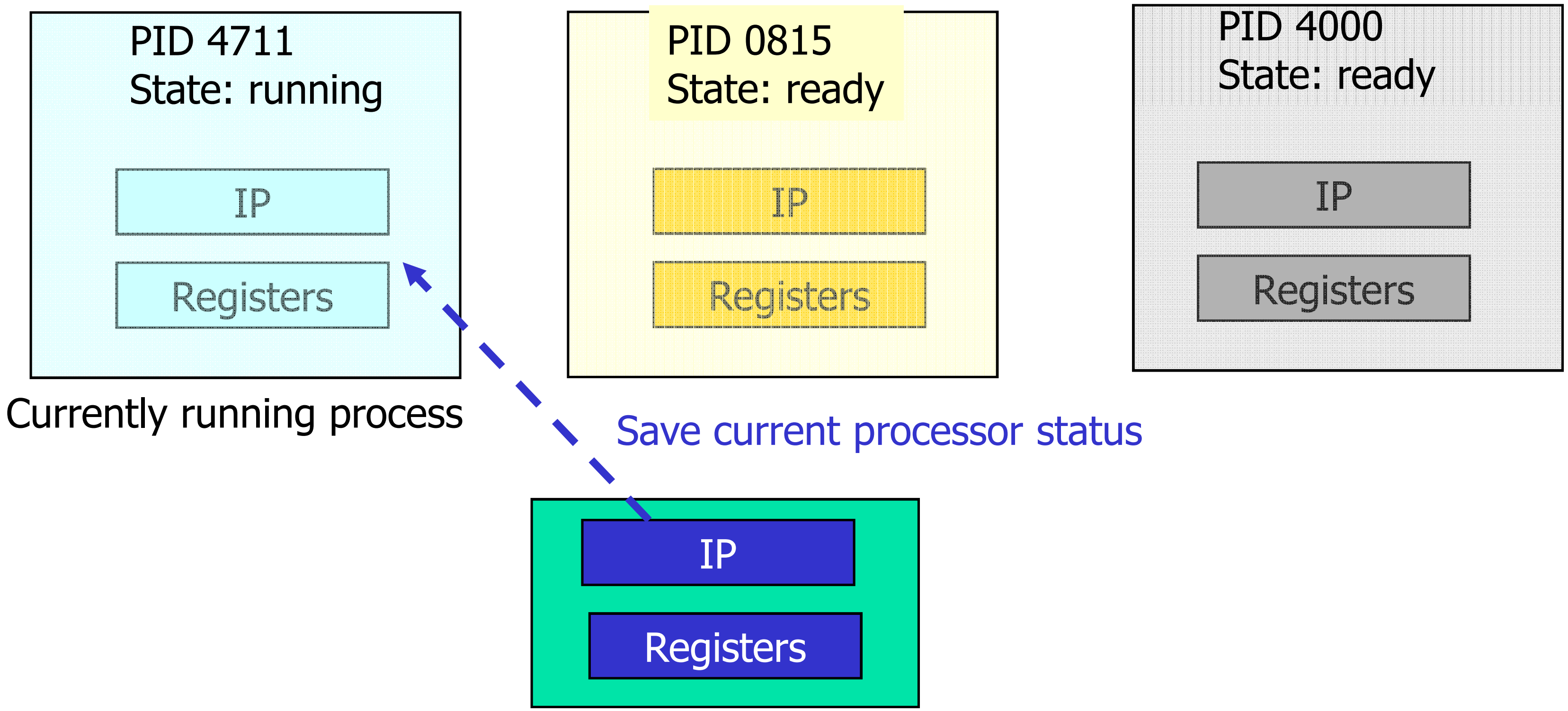


Overhead to create a Process

- Must construct a new PCB
  - Quite cheap (as long as there is enough space for it)
- Must set up new page tables or related data structures representing the new AS
  - More expensive
- Copy data from parent process? (Unix `fork()`)
  - Semantics of Unix **`fork()`**: the child process gets a complete copy of the parent’s memory and I/O state
  - In early Unix versions very expensive
  - Today less expensive due to concept of **“**copy on write**”**
- Copy I/O state (file handles, etc)
  - More expensive

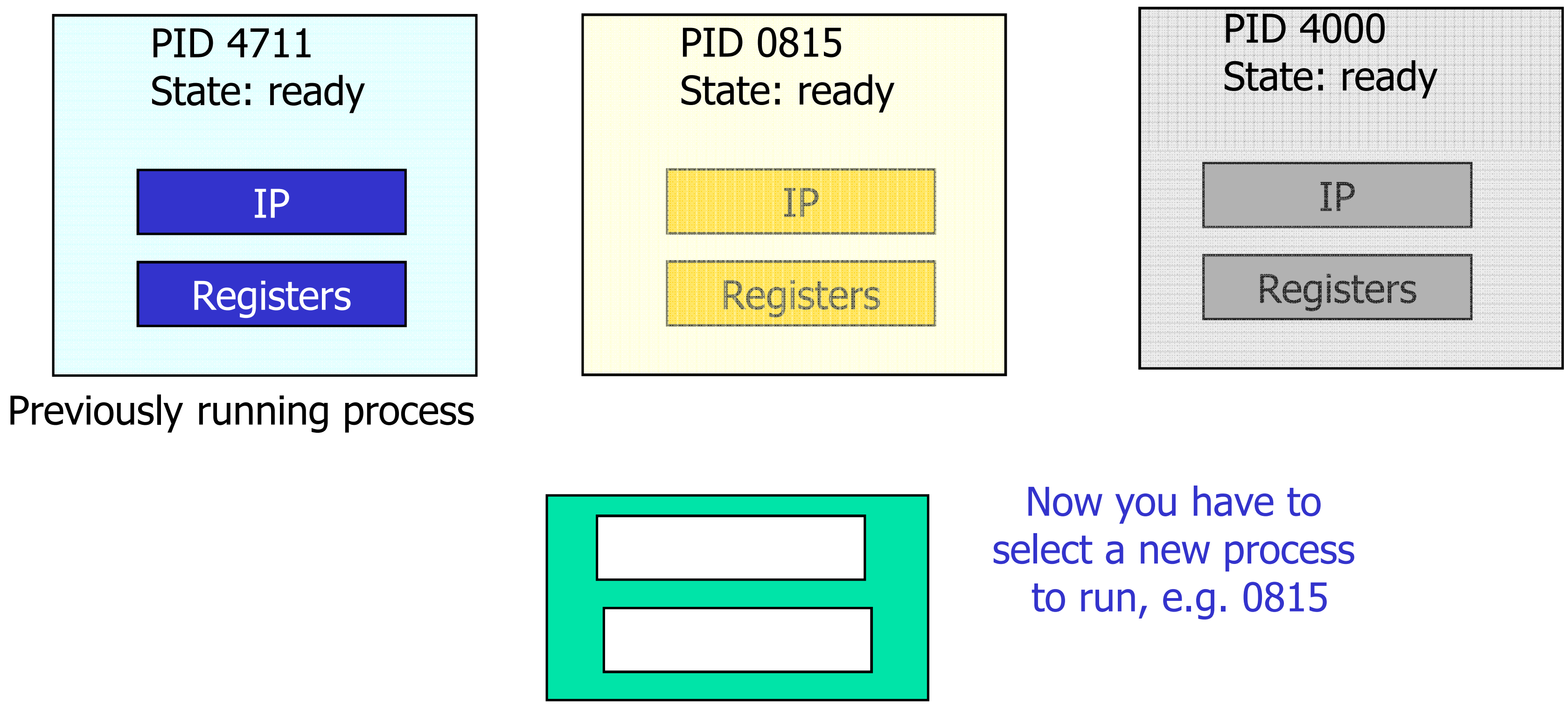
Preview: Process Switching

- Action of releasing one process from the CPU and assigning another process to the CPU is named a voluntary process switch



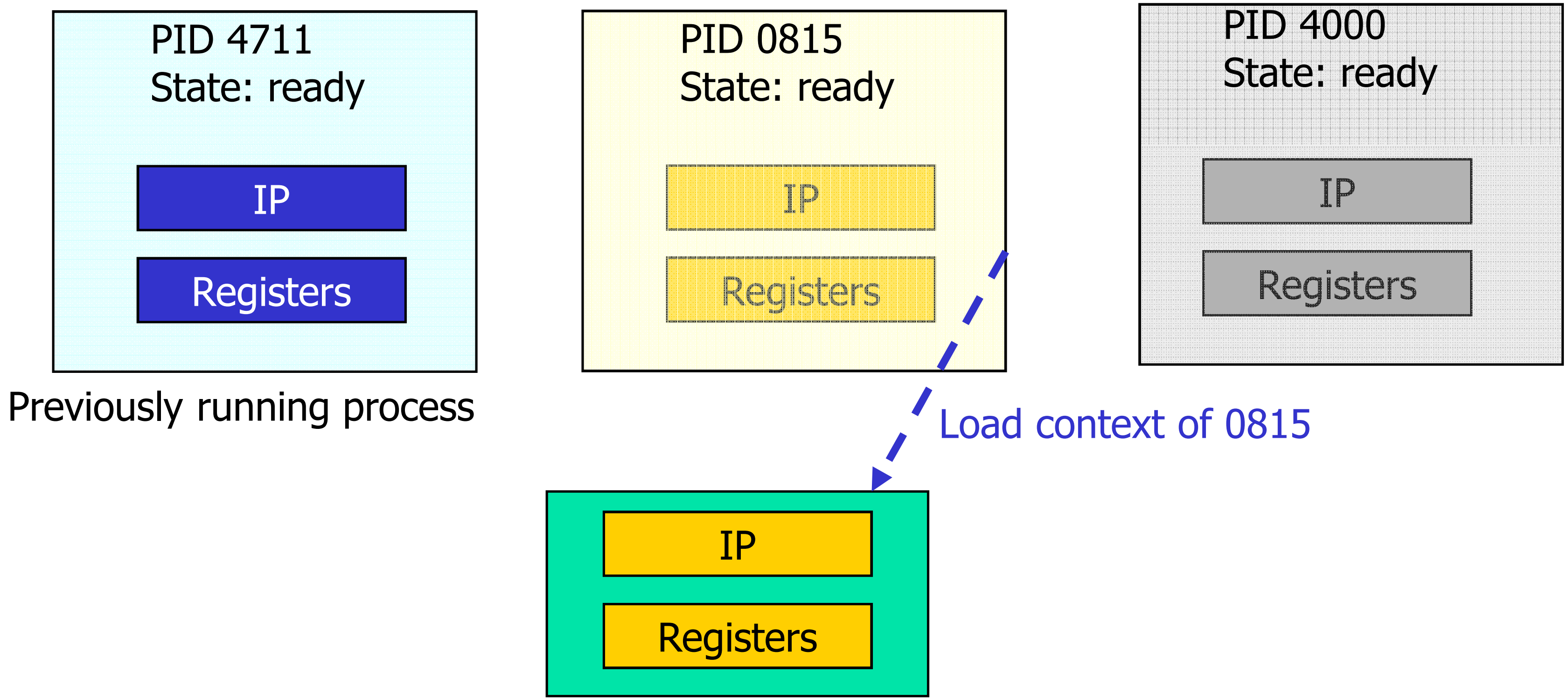
Process Switching

- Intermediate state



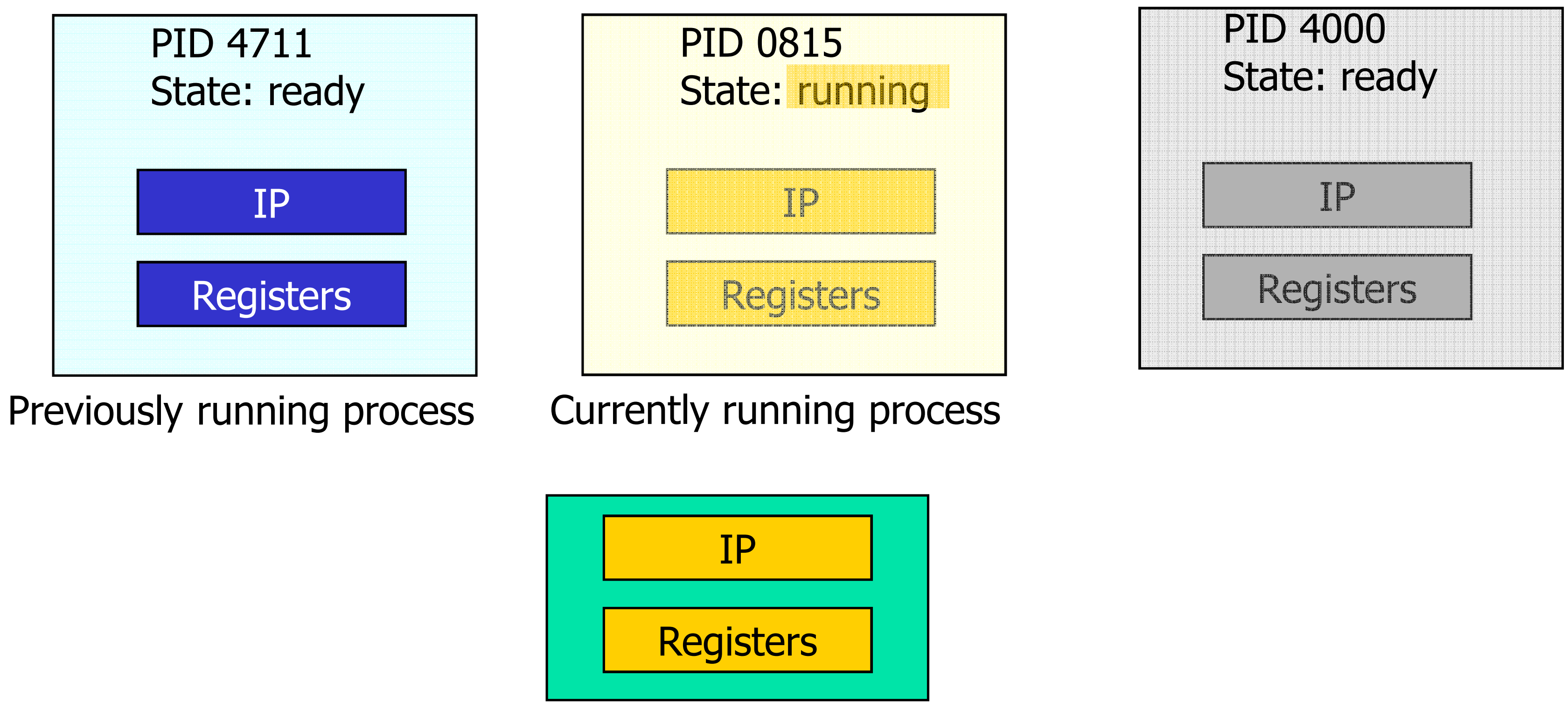
Process Switching

- Loading context of the new process



Process Switching

- Running the new process

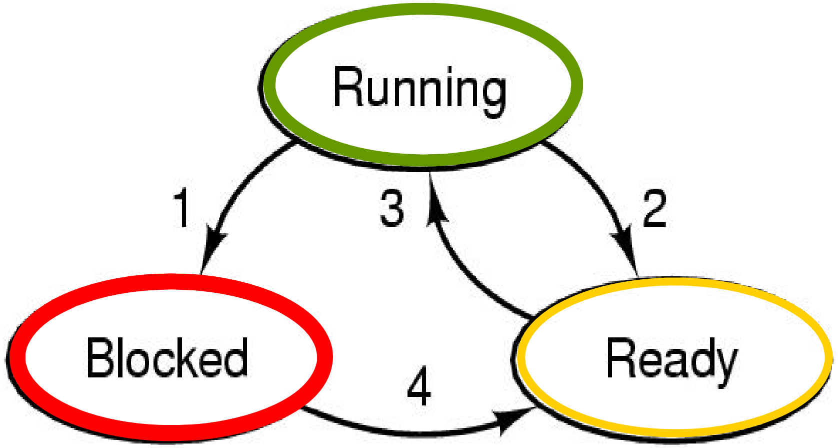


Cost of Process Switching

- In some systems, a process switch is expensive
  - Entering and exiting the kernel
  - CPU context has to be saved & restored
  - Storing & loading AS information
  - (Flushing TLB and) restoring TLB content
  - Scheduling next ready process to run can be expensive, especially when the OS designer is too lazy
- Process switch overhead in Linux 2.4.21
  - ~ 5.4 µsec on a 2.4 GHz Pentium 4
  - Equivalent to ~ 13 200 CPU cycles !!!
  - Not quite that many instructions since CPI* >1

*CPI = cycles per instruction

(External) Process States



1. Process blocks for I/O
2. Process leaves CPU voluntarily or scheduler forces process to leave the CPU
3. Scheduler picks another process
4. I/O has occurred, process no longer has to wait

Possible process states

- **Blocked** (waiting, sleeping)
- **Ready**
- **Running**

Colors should remind you of the semantics of traffic lights

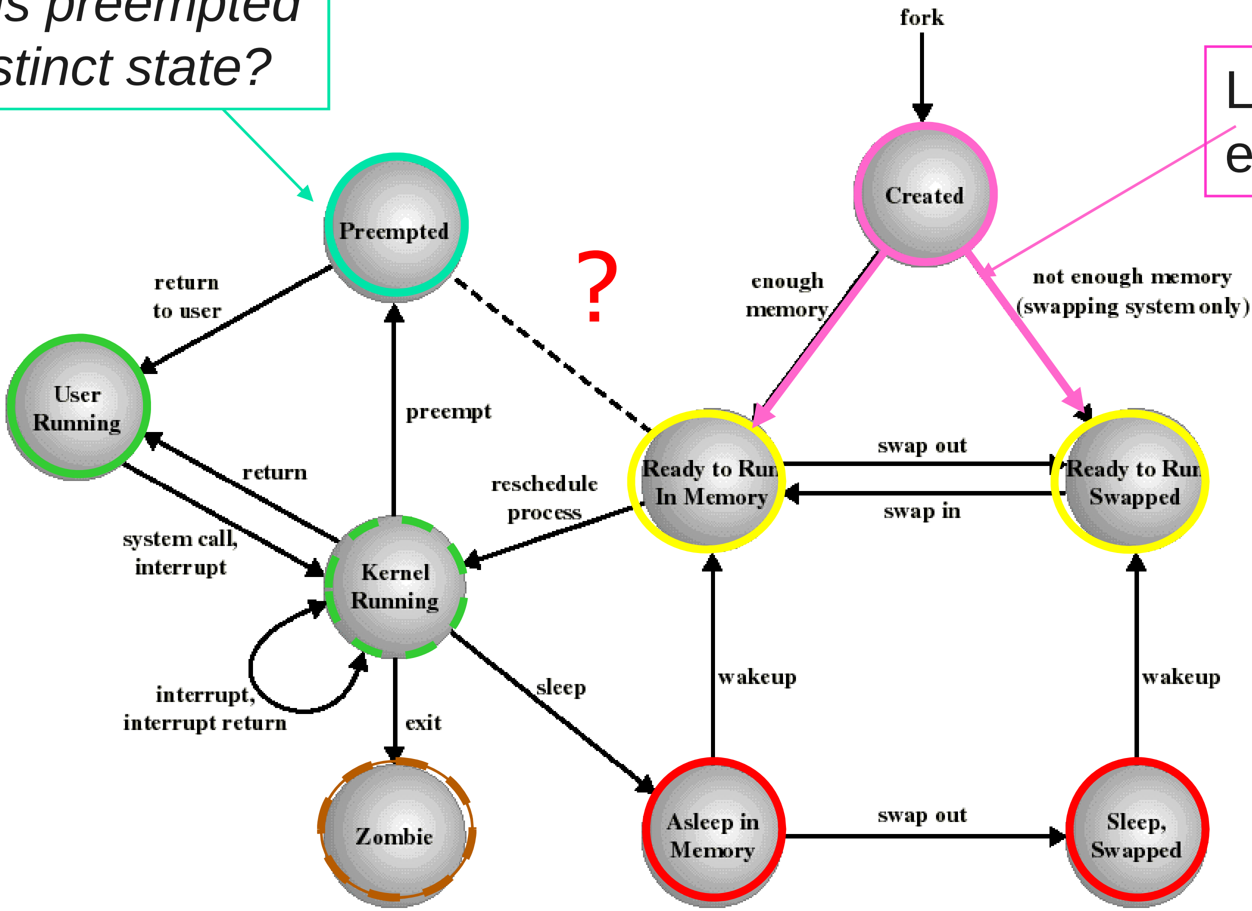
Process States

- During its life a process can change its external process state several times
  - new: Process has being created
  - running: Its instructions are executed
  - waiting: It is waiting for some internal or external event to occur
  - ready: It is ready to run, however it is still waiting to be assigned to a processor
  - terminated: It has finished

Unix "Process State Model"

Why is preempted a distinct state?

Lazy versus eager loading



Potential Attributes of a PCB

	Process management	Memory management	File management
Context	Registers Program counter Program status word Stack pointer	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID
Scheduling	Process state Priority Scheduling parameters		
Family	Process ID Parent process Process group		
Time & Events	Signals Time when process started CPU time used Children's CPU time Time of next alarm		

Shortcomings of Process Model

- Sufficient for all sequential applications
  - e.g. only one activity per process
- *However, what to do if your application can profit from internal concurrency?*

⇒ Multiple application processes
  - Protection is guaranteed by different AS, but solution can be expensive
  - Collaboration takes time

- Better use a multi-threaded task
  - ∃ different thread models

Example Parallel Applications

- Web browser:
  - Download web pages, read cache files, accept user input,…
- Web server:
  - Handle incoming connections from multiple clients
- Scientific programs:
  - Process different parts of a data set on different CPUs, e.g. calculate a numerical difference equation

Parallel Applications

- Share memory across multiple activities
  - Web browser: share buffer for HTML pages
  - Web server: share memory cache of recently accessed pages
  - Scientific programs: share memory of global data set being processed
- Can we do this with multiple processes?
  - Yes, as long as OS offers shared memory
  - If not, we must use IPC which might be inefficient

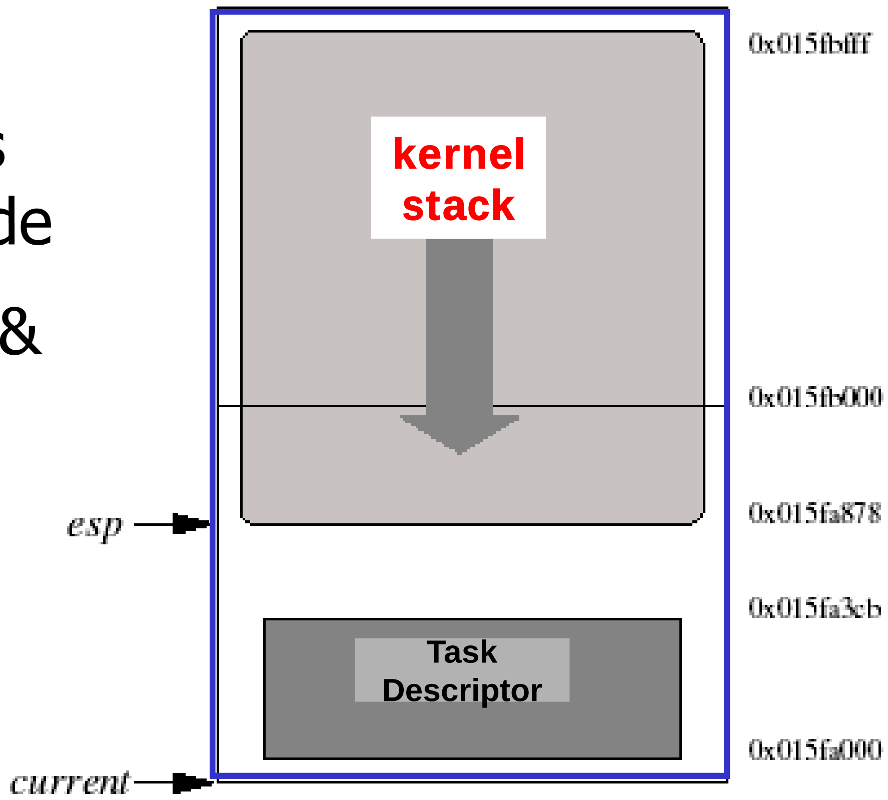
Task Model

Example: Linux T(ask)CB

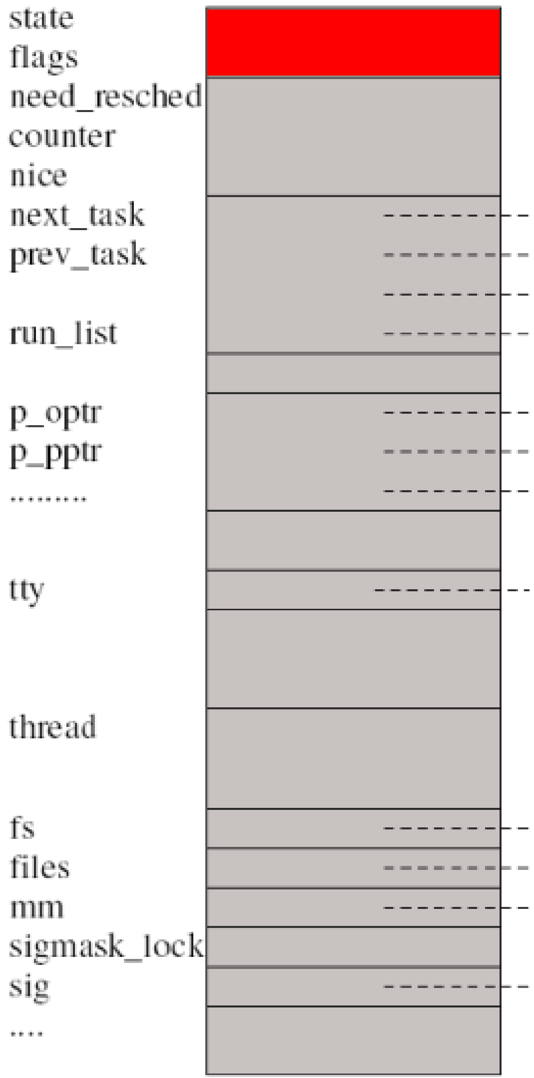
Linux Task Descriptor TaskCB

- Located in a 8 KB kernel memory block
- esp register points to kernel stack if task has switched to kernel mode
- The macros **`current`** & **`esp`** deliver the task-descriptor address

Example TaskCB



Content of Task Desciptor (1)



State:

- current task state

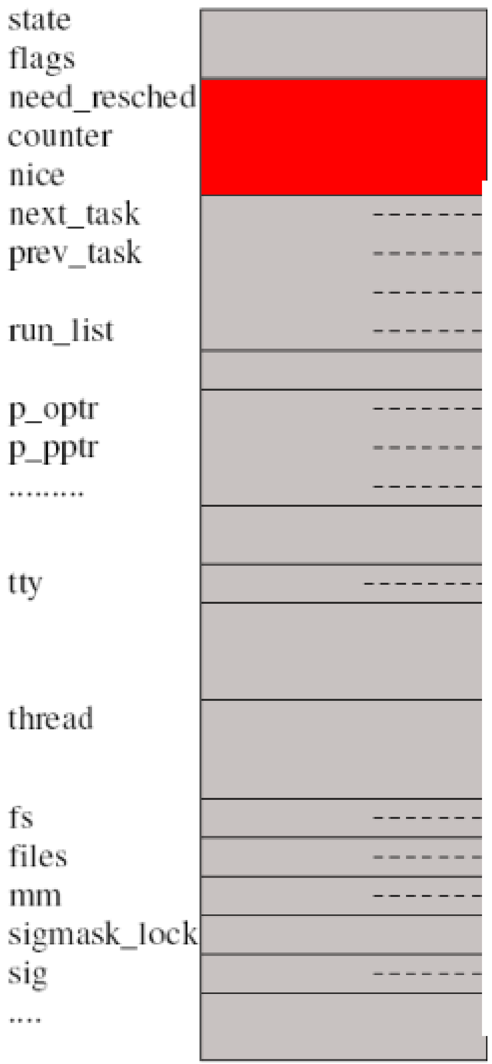
Macros for changing the task-state by the kernel:

- set_task_state(TID) = set the state of a certain task with TID
- set_current_state = set the state of a the running task

Flags:

- part of PSW or Statusregister

Content of Task Descriptor (2)



counter:

- Time in ticks (10 ms) till next scheduling
- Used to select next running task (thread)
- Dynamic priority

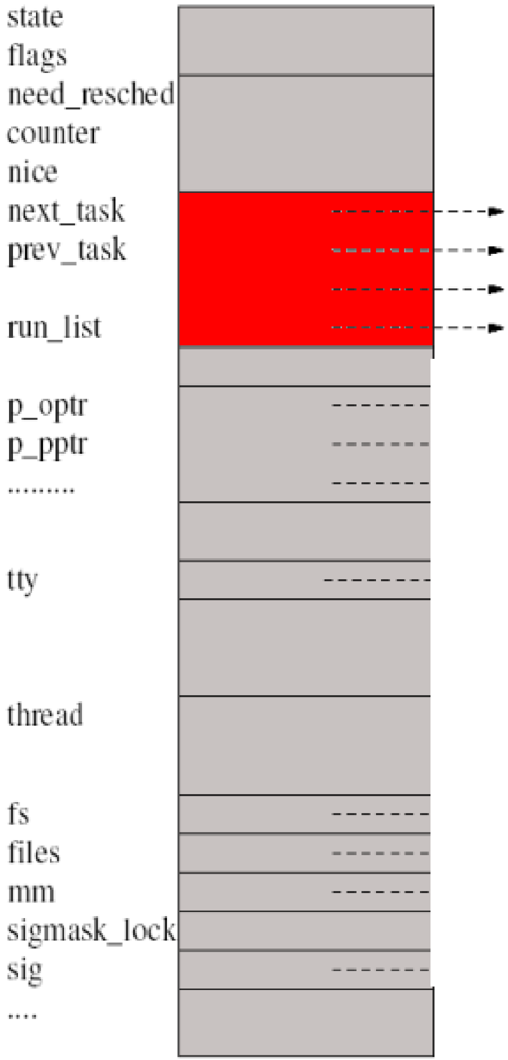
nice(priority):

- Static priority
- Scheduler uses priority to set counter

need_resched:

- Indicates that scheduling has to be done (sooner or later)

Content of Task Descriptor (3)

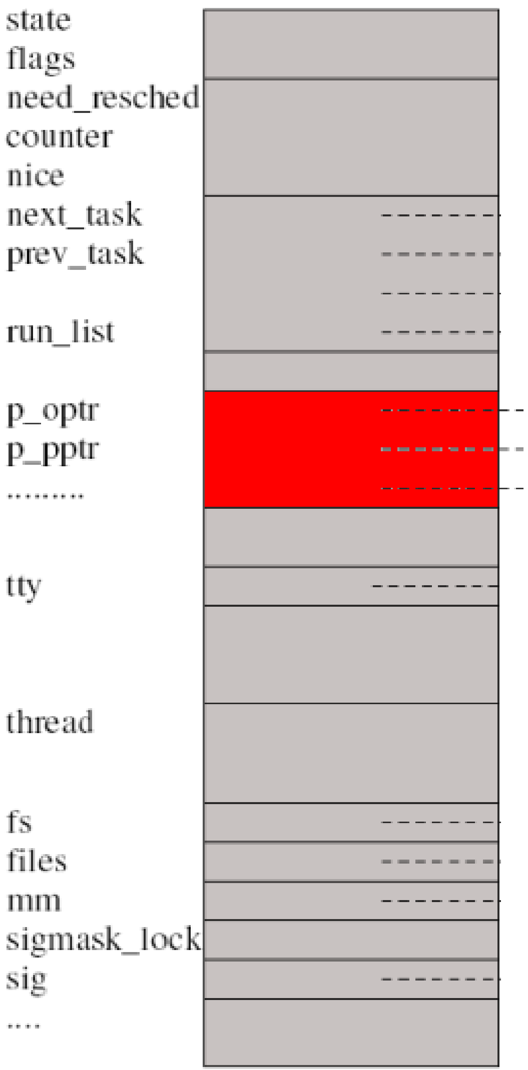


Next_task: next task

Prev_task: previous task

Run_list: list of ready tasks

Content of Task Descriptor (4)

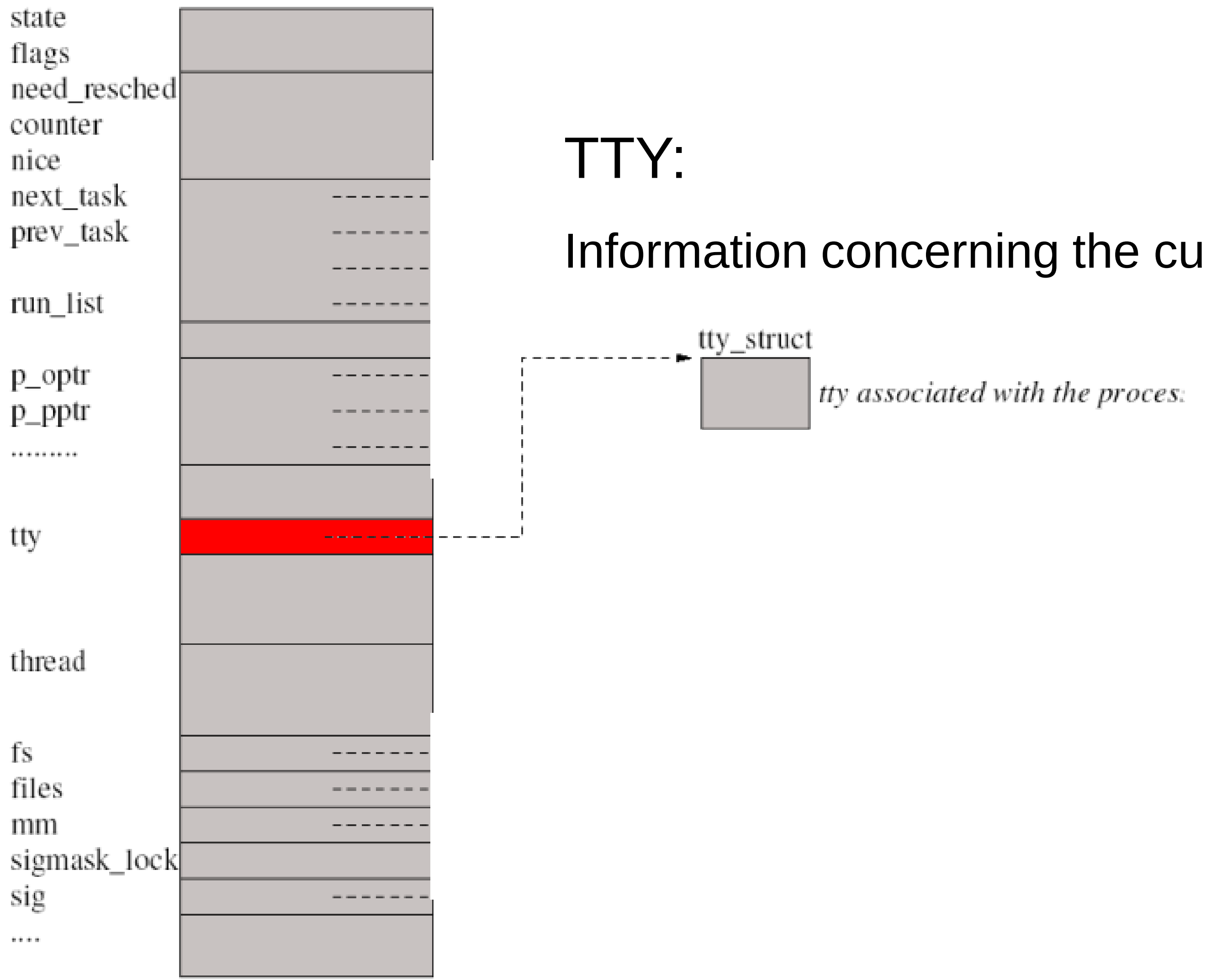


p_opptr, p_pptr, p_cptr,

p_ysptr; p_osptr::

- e.g. old or current parent etc.

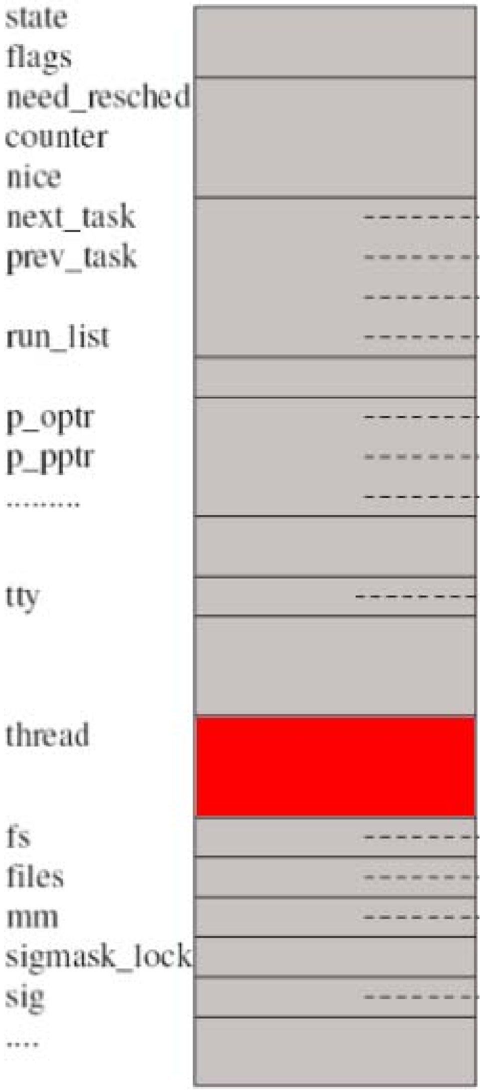
Content of Task Descriptor (5)



TTY:

Information concerning the current console

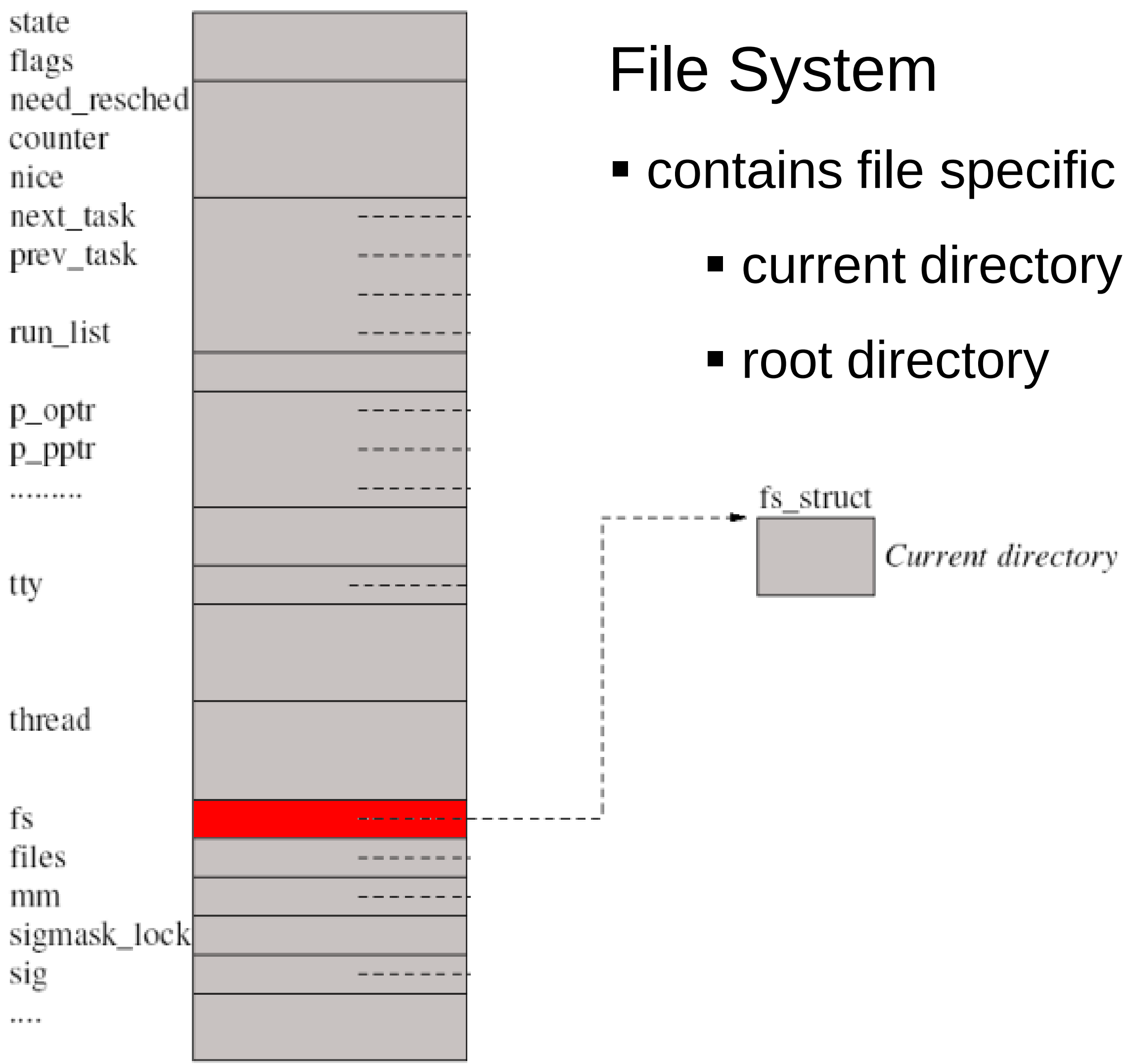
Content of Task Descriptor (6)



Thread:

- Information about CPU state for last mode switch
- Save all CPU registers

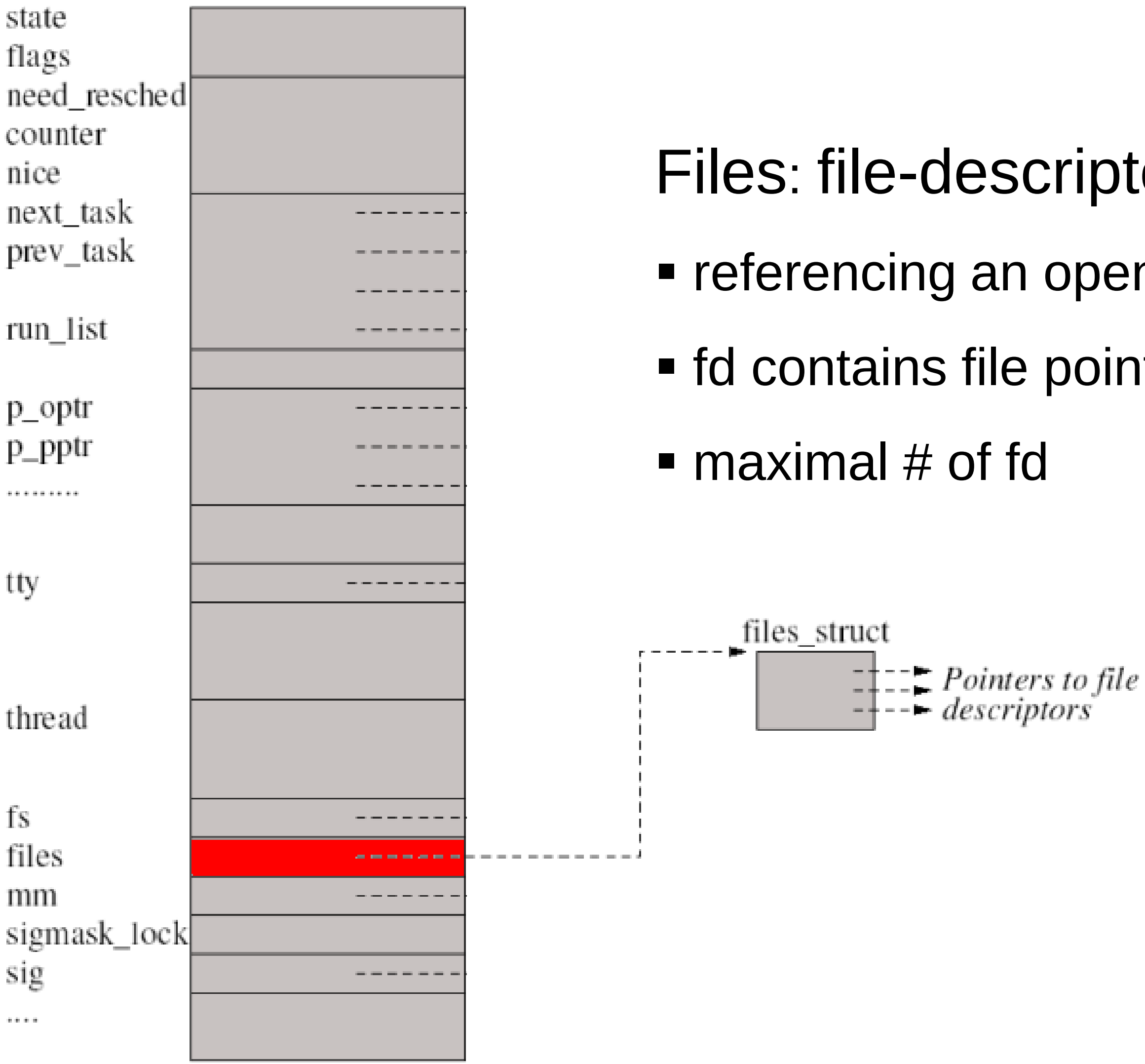
Content of Task Descriptor (7)



File System

- contains file specific information:
  - current directory
  - root directory

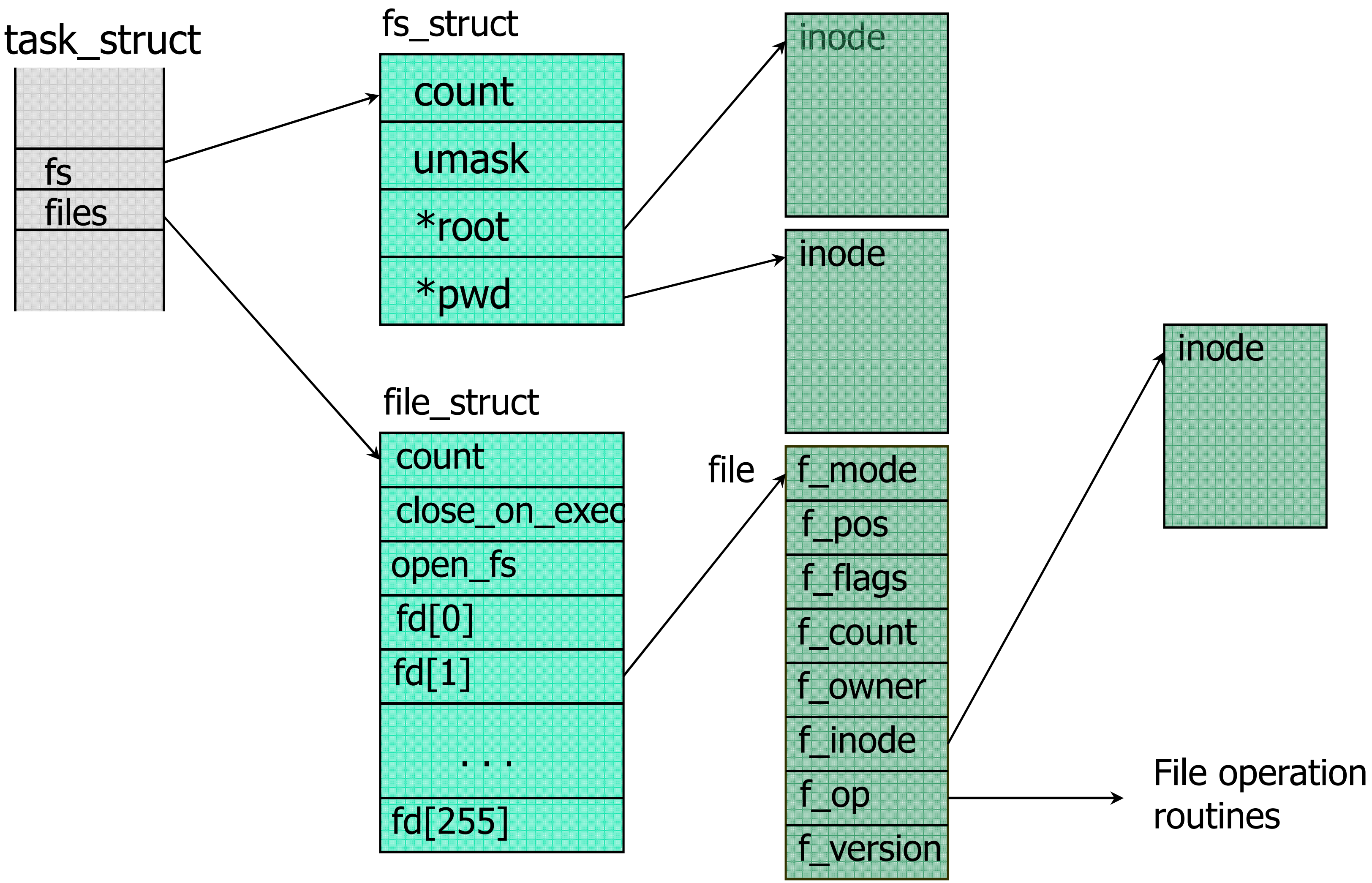
Content of Task Descriptor (8)



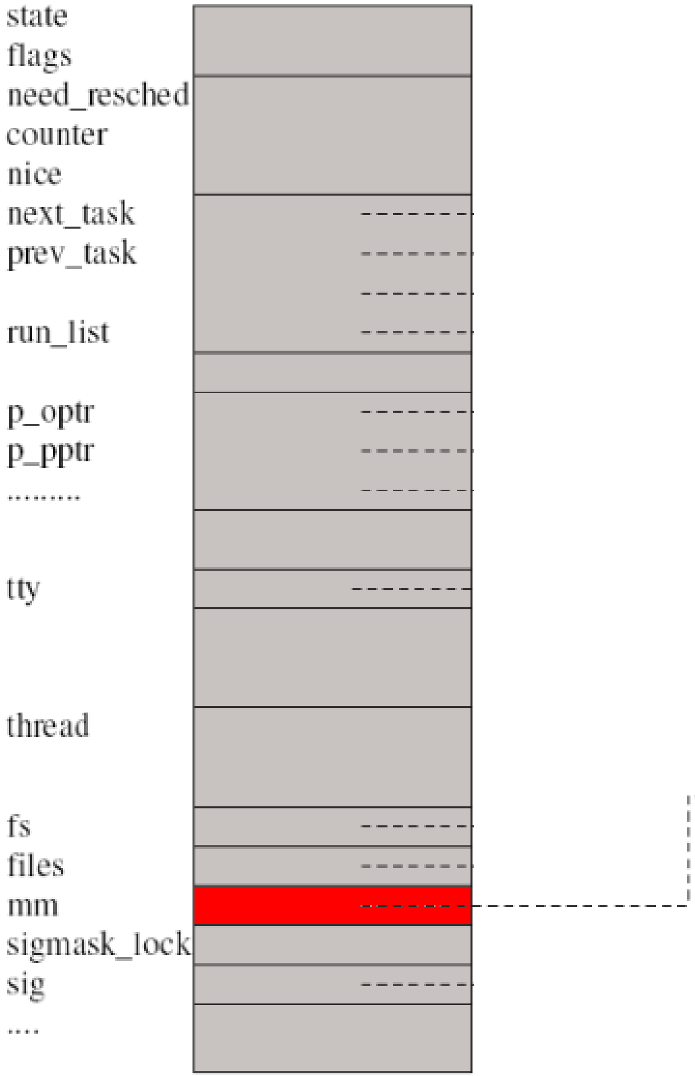
Files: file-descriptor

- referencing an open file
- fd contains file pointer
- maximal # of fd

Content of Task Descr. (7,8 a)

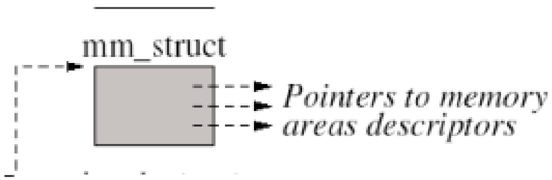


Content of Task Descriptor (9)

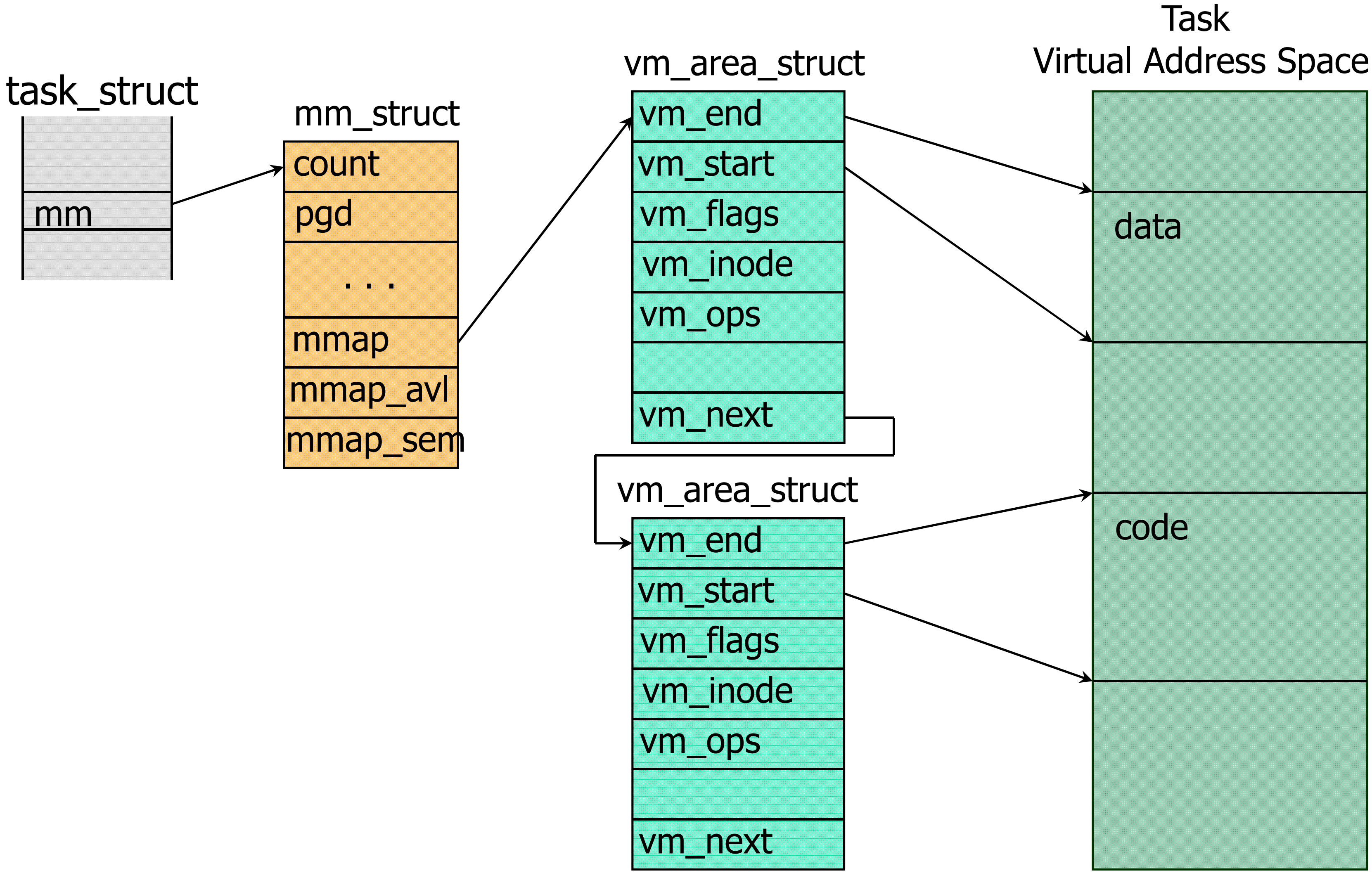


mm: memory management

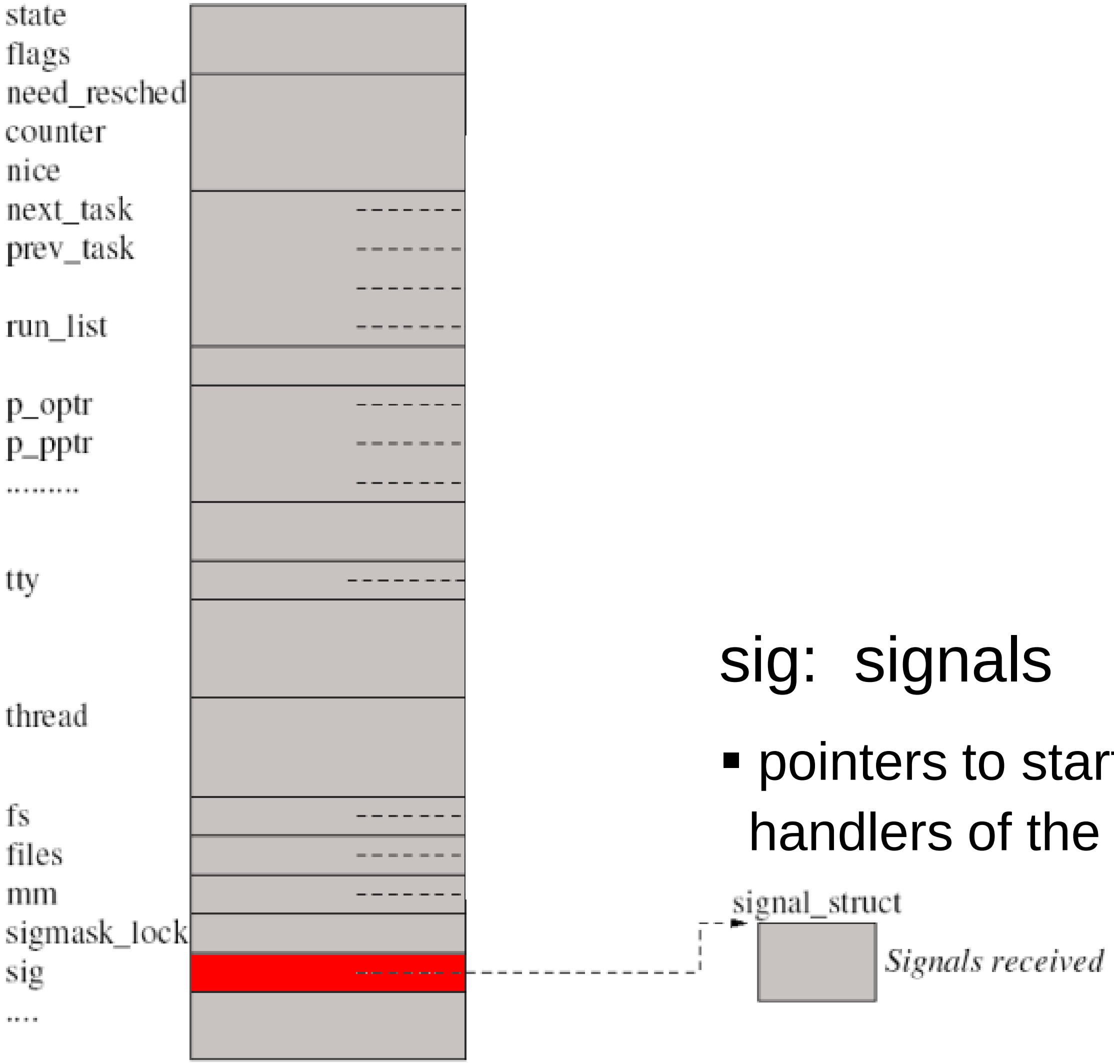
- contains pointers to memory areas



Content of Task Descriptor (9a)



Content of Task Descriptor (10)



sig: signals

- pointers to start-addresses of signal handlers of the task

Thread Model

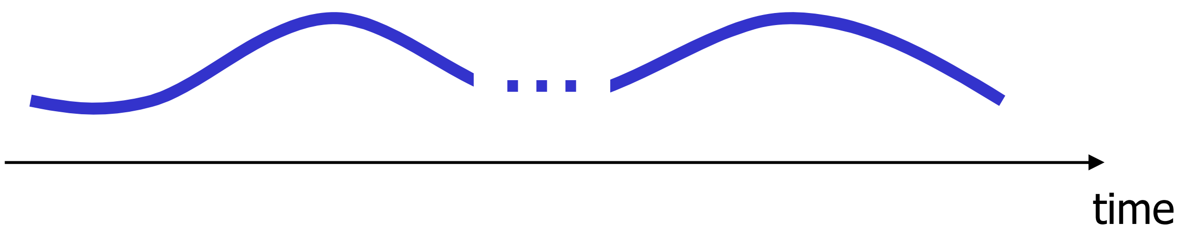
Thread

- Thread = abstraction for a **pure activity**
- Thread includes **code** and **private data** (stack)
- Each thread needs an ***execution environment***
  - Address space
  - Files, I/O-devices and other resources
  - In many cases, a thread shares its complete environment with all other threads of the same AS

Example: File server consists of **t** identical threads, each thread serves one client's request

Thread

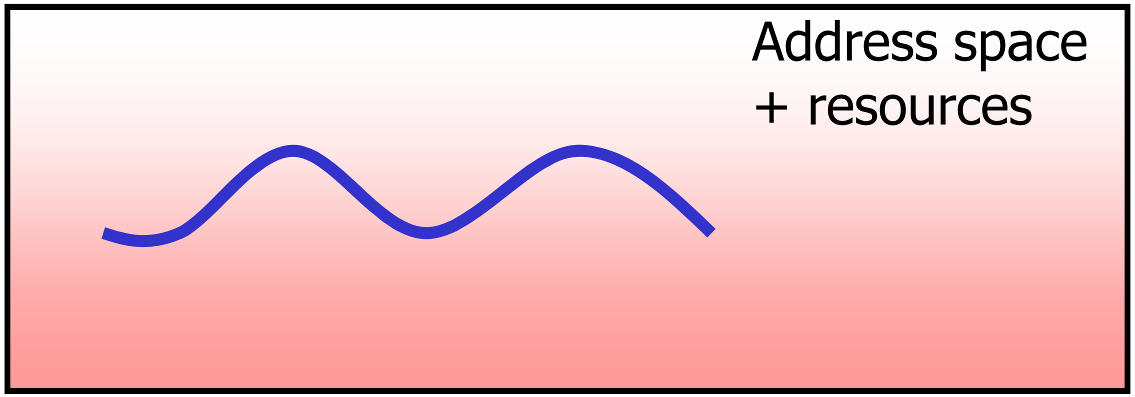
- Entity in which activity takes place
- Object of dispatching (scheduling)



On a single processor system, threads are executed on the same CPU, thus we need to control all threads in order to prevent that a single thread is hogging the CPU

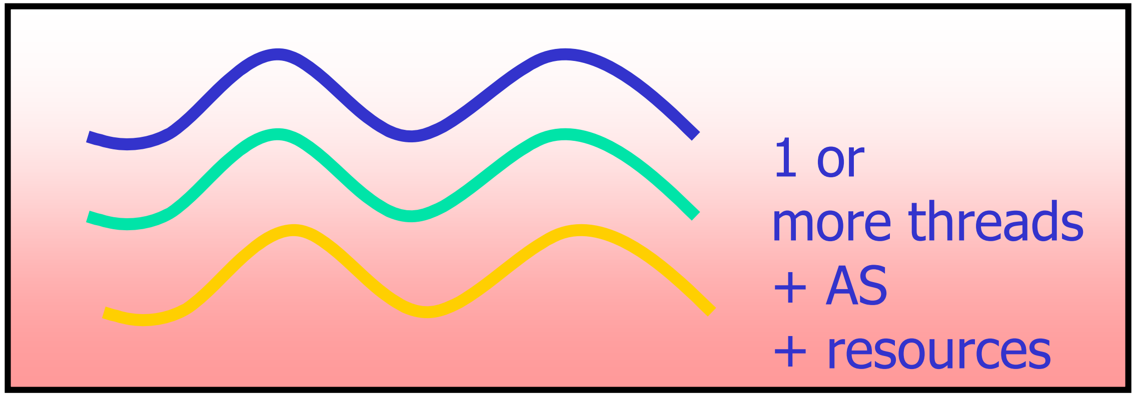
Process

- Single threaded
- Address space (Unix terminology)
- Additional resources



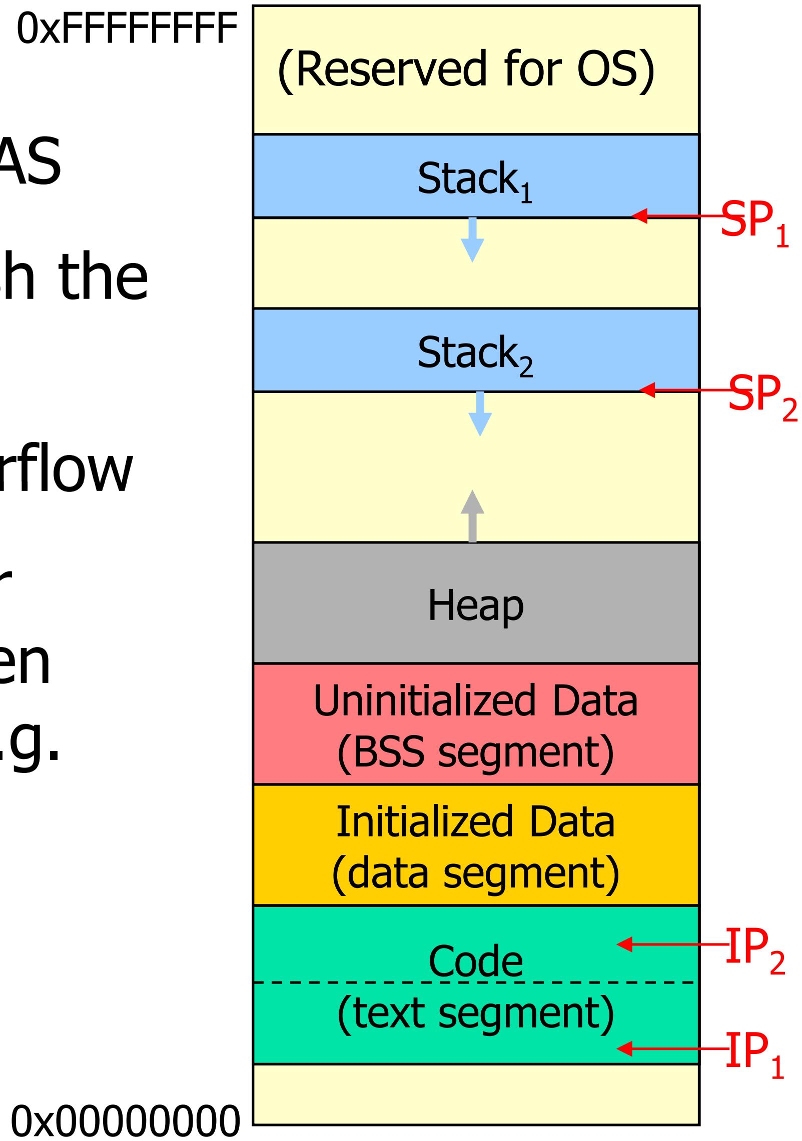
Task

- Entity of an "application" consisting of
  - t ≥ 1 thread(s)
  - Address space
  - Resources

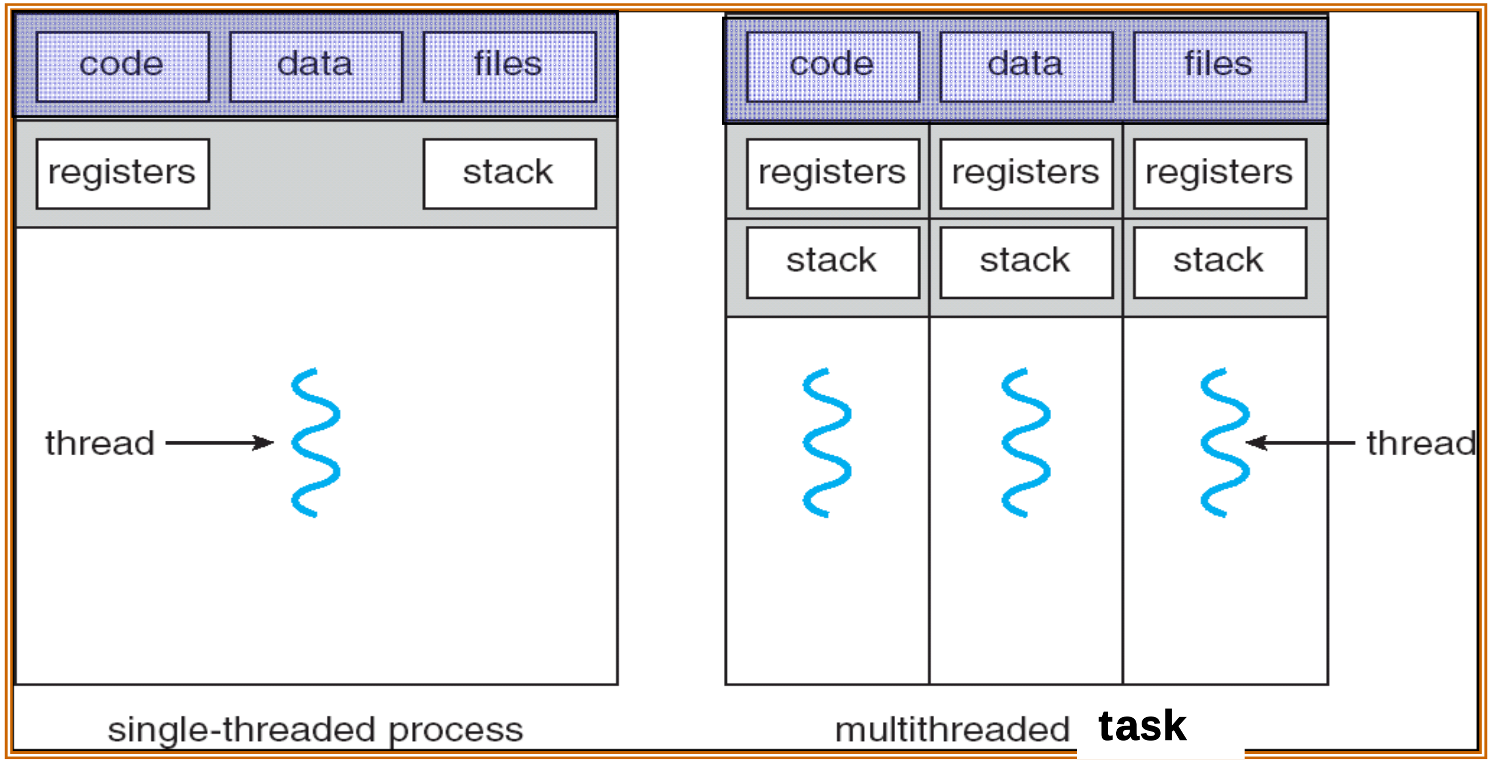


Address Space of a Task

- All threads share the same AS
- Bugs in one thread can crash the complete task
- Danger of mutual stack overflow
- Robust systems should offer additional protection between threads of the same task, e.g.
  - Private global data
  - Protected thread stacks



Process versus Task



- Threads encapsulate concurrency: “active” component
- Address spaces encapsulate protection: “passive” part
  - Keeps buggy process from trashing other processes or the system

Example Multithreaded Programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Modern OS kernels
  - Contains kernel threads (no kernel level threads)
    - Kernel threads are completely executed in kernel mode
    - Kernel level threads are executed in user mode most of the time
  - Often, no/few additional protection offered inside the kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

Literature

Bacon, J.: Operating Systems (4)

Stallings, W.: Operating Systems (3, 4)

Silberschatz, A.: OS Concepts (2)

Tanenbaum, A.: MOS (2)