Operating Systems

Processes

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

Process

A process is a program being executed.

In a multitasking OS, multiple instances of the same program might be concurrently running.

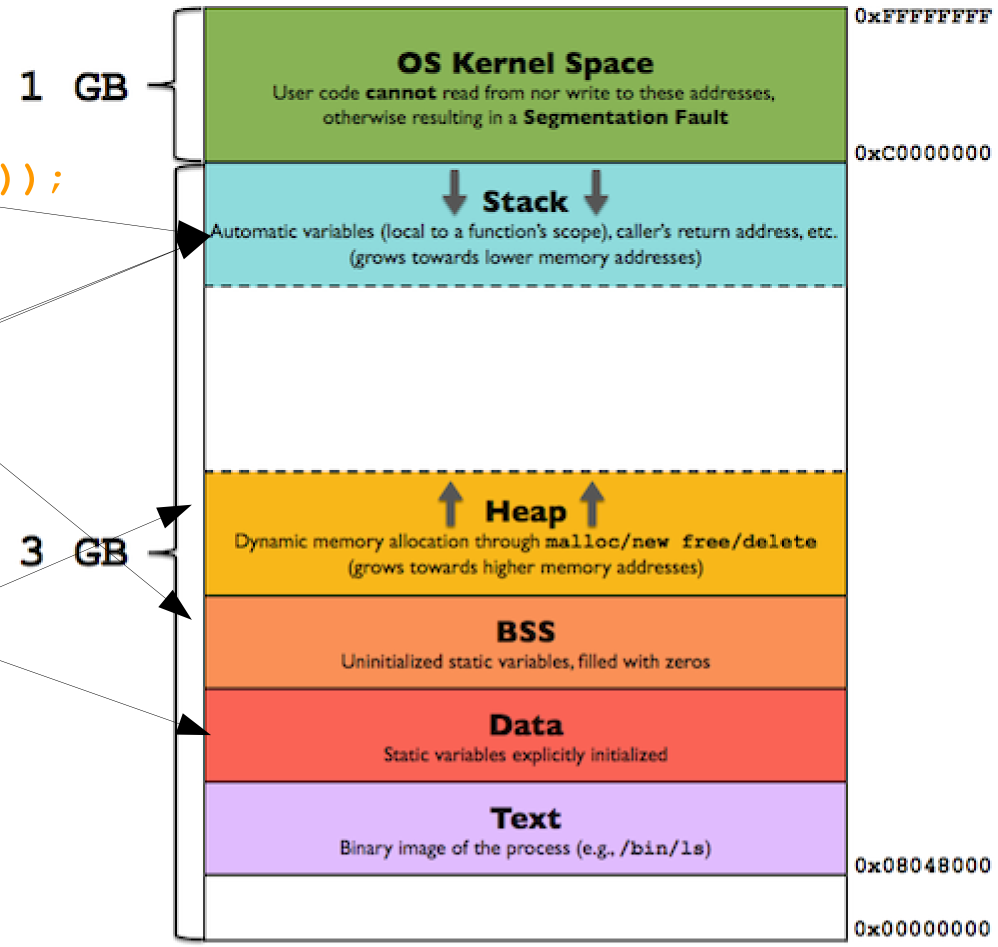
Process is characterized by its state:

- CPU registers
- Memory
  - code(.text)
  - stack
  - heap
  - global variables (.bss and .data)
  - memory mapped regions (between stack and heap)
- Resources
  - file/socket descriptors
  - synchronization constructs (semaphores, queues)

Process Memory

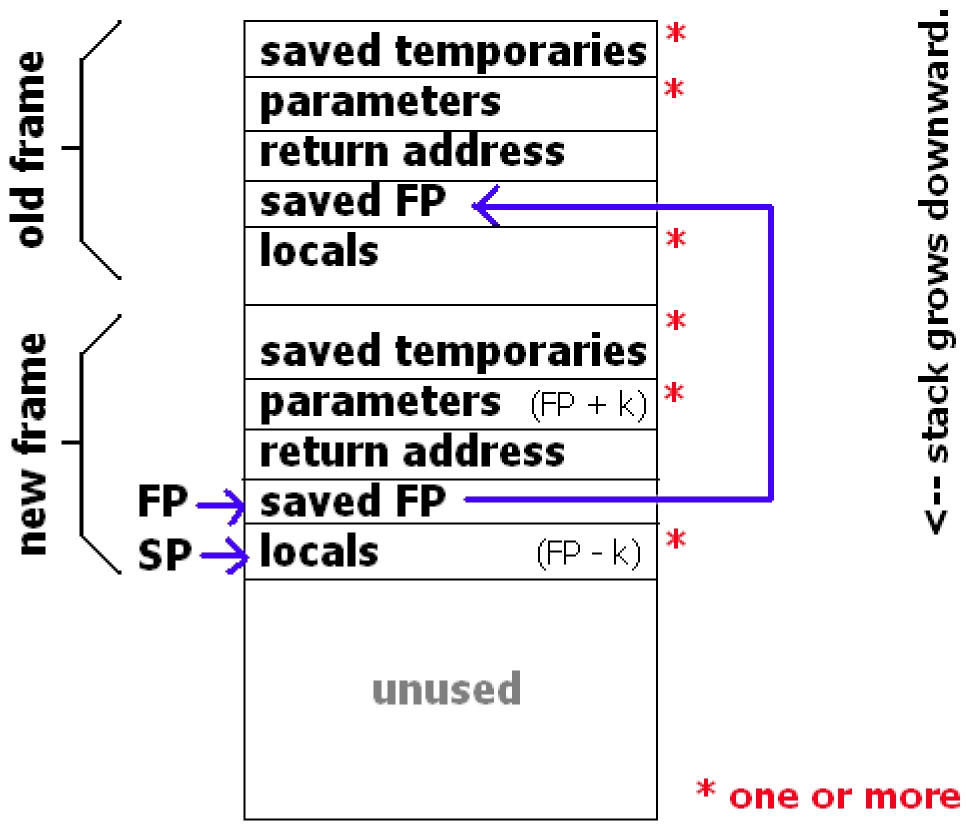
```
int g_counter;

int f(int n)
{
    float *res;
    res = (float *)calloc(n, sizeof(float));
    /* .....*/
    free(res);
    g_counter++;
}
int main()
{
    char *str = "ciao";
    int vect[1000];
    int *p;
    p  = (int *)malloc(10*sizeof(int));
    /* ... */
    free(p);
    g_counter++;
}
```

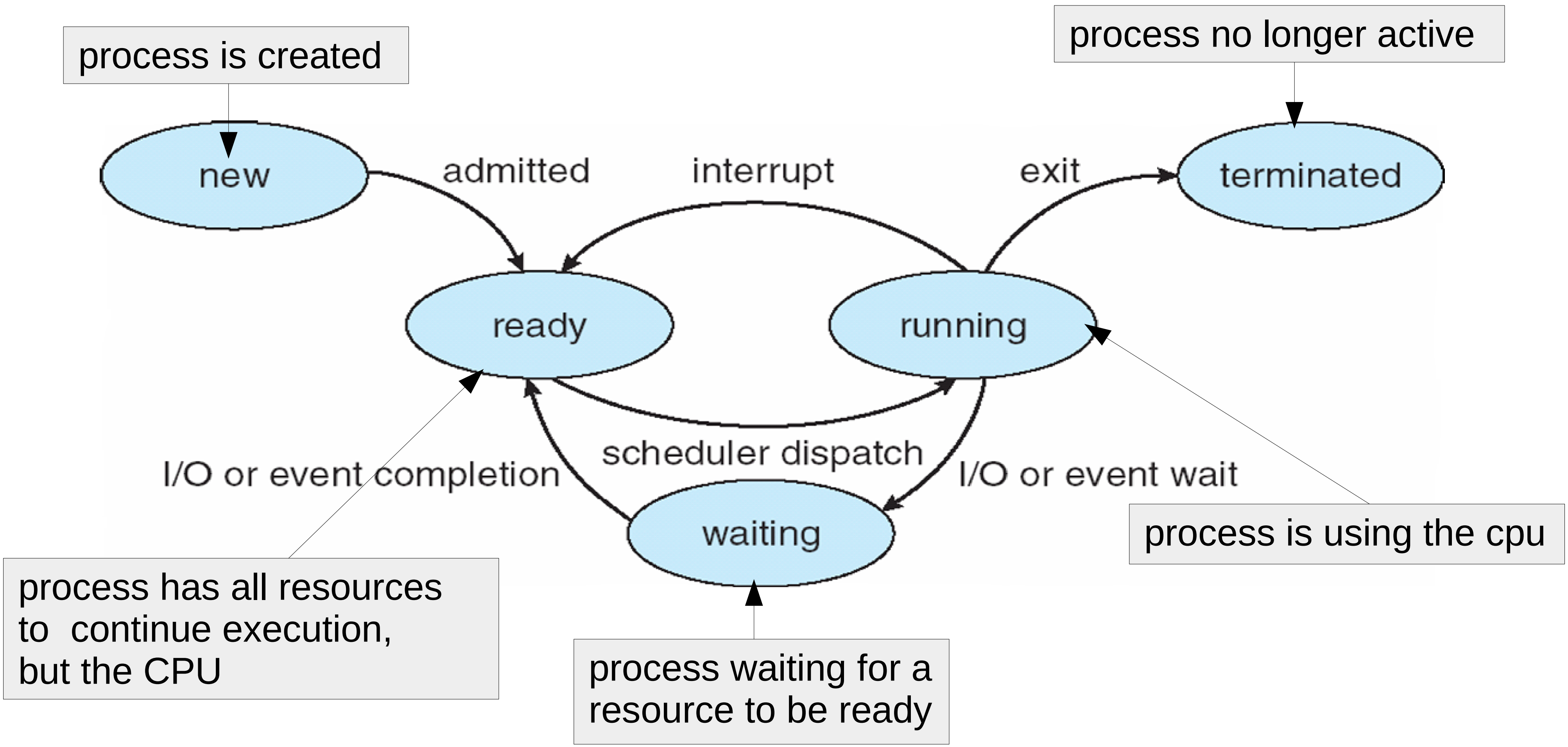


Stack Frame

- Pointed by a special register
- Stores the activation records for each function
  - arguments
  - return address
  - saved (clobbered) registers
  - local variables



Process States



Unix Like Process Control

- Create a new process
- Wait for a process to terminate
- Load a process file, and execute it
- Terminate a process
- Handle an asynchronous event

fork()

System call used to create a new process

After fork two instances of the same process are created:

- memory is "copied" from creator (parent) to the created (child)
- file descriptors and resources are copied too
- the return value of fork is 0, for the child, **`child_pid`** for the parent
- use getpid() to get pid of a process (a unique identifier for a process in a system);

```
// typical fork example

int main(int argc, char** argv) {
     int v=fork();
     if (v){
          // we are in the parent;
          doParentStuff();

          // see in 2 slides
          // waits for child termination;
          int retval;
          int pid=wait(&retval)
     } else {
          doChildStuff();
          return 0
     }
}
```

wait / waitpid

Suspends the execution of a process until

- one of his child processes terminate (wait)
- a specific child process terminates(waitpid)

```
int main(int argc, char** argv) {
     int v=fork();
     if (v){
          // we are in the parent;
          doParentStuff();

          // see in 2 slides
          // waits for child termination;
          int retval;
          int pid=wait(&retval)
     } else {
          doChildStuff();
          return 0;
     }
}
```

Parent and Childs

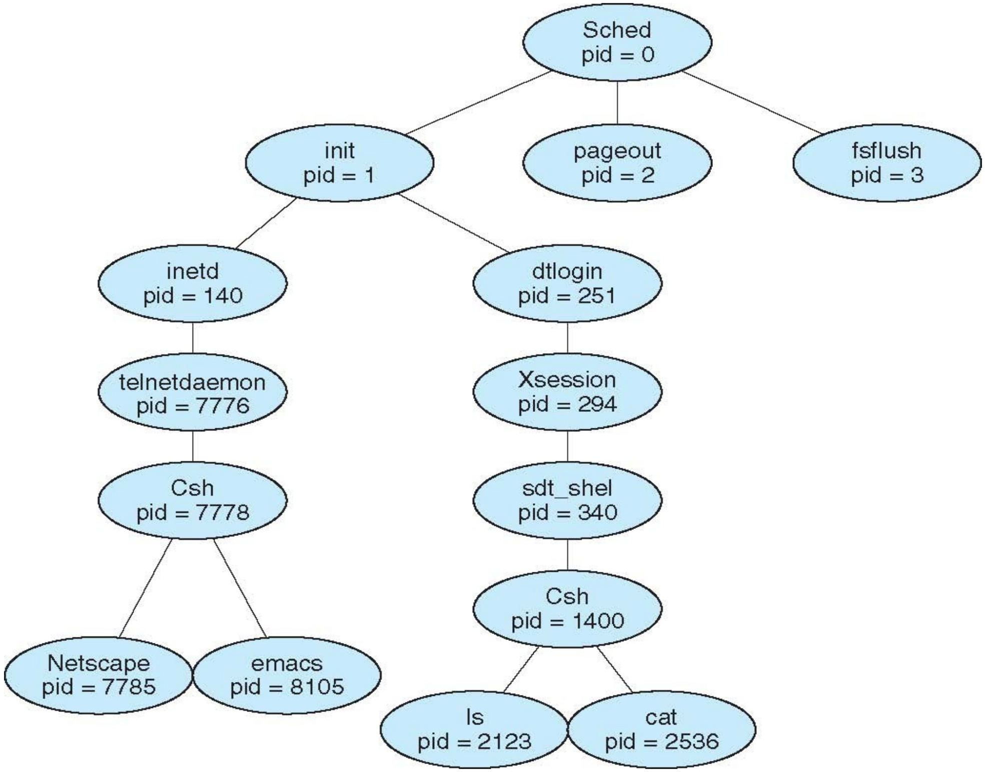
The creator/created relation is naturally captured by a process tree

Issue:

- What happens when a parent process terminates before his child?

Solution:

- The orphan process becomes child of the mother of all processes (init/systemd), that is the first process started by the system.



fork/wait example

Spawns two processes: the parent and the child

Both execute a useless loop for a certain number of rounds

(see parent and child)

One of the two will die earlier.

What happens if the father dies earlier than the child?

What happens if the child dies earlier than the parent?

```
const char parent_prefix[]="parent";
const char child_prefix[]="child";
const char* prefix=parent_prefix;
pid_t pid;
const int num_rounds_parent=10;
const int num_rounds_child=5;

void childFunction() {
  for (int r=0; r<num_rounds_child; ++r) {
    printf("%s looping, pid: %d, round: %d \n",
           prefix, pid, r);  sleep(1); }
}
void parentFunction() {
  for (int r=0; r<num_rounds_parent; ++r) {
    printf("%s looping, pid: %d, round: %d \n",
          prefix, pid, r); sleep(1);  }
}
int main(int argc, char** argv) {
  pid=getpid(); // here we store the process id
  printf("%s started, pid: %d\n", prefix, pid);
  printf("%s now forking\n", prefix);
  pid_t fork_result=fork();
  if(fork_result==0){
    prefix=child_prefix;
    pid=getpid();
    printf("%s started, pid: %d\n", prefix, pid);
    childFunction();
  } else {
    printf("%s continuing, pid: %d\n", prefix, pid);
    parentFunction();
  }
  printf("%s terminating, pid: %d\n", prefix, pid);
}
```

On Termination

A parent process is notified of the termination of one of his childs, by the OS.

When a child process dies the OS sends the parent a SIGNAL (SIGCHLD)

When a parent terminates, all his alive children are notified about the termination through another signal (SIGHUP)*

Signal handlers can be installed through the signal(...) od sigaction(...) syscalls, that takes:

- the signal number
- a function pointer to the handler

***not by default on linux**

Example of SIGCHLD/SIGHUP

```
void sigchld_handler(int signal) {
  printf("SIGNAL %s got signal %d,
          child is dead\n", prefix, signal);}

void sighup_handler(int signal) {
  printf("SIGNAL %s got signal %d,
          parent is dead\n", prefix, signal);}

int main(int argc, char** argv) {
  pid=getpid(); // here we store the process id
  printf("%s started, pid: %d\n", prefix, pid);
  pid_t fork_result=fork();
  if(fork_result==0){
    prefix=child_prefix;
    pid=getpid();
    printf("%s started, pid: %d\n", prefix, pid);
    struct sigaction new_action, old_action;
    new_action.sa_handler = sighup_handler;
    sigemptyset (&new_action.sa_mask);
    new_action.sa_flags = 0;
    sigaction (SIGHUP, NULL, &old_action);
    if (old_action.sa_handler != SIG_IGN) {
      sigaction (SIGHUP, &new_action, NULL);
    } else {
      exit(-1); // error
    }
    childFunction();
  }
  else {
    struct sigaction new_action, old_action;

    new_action.sa_handler = sigchld_handler;
    sigemptyset (&new_action.sa_mask);
    new_action.sa_flags = 0;
    sigaction (SIGCHLD, NULL, &old_action);
    if (old_action.sa_handler != SIG_IGN) {
      sigaction (SIGCHLD, &new_action, NULL);
    } else {
      //error
      exit(-1);
    }
    printf("%s cont, pid: %d\n", prefix, pid);
    parentFunction();
    printf("%s sending sighup to %d\n",
           prefix, fork_result);
    // on linux we should explicitly raise sighup
    kill(fork_result, SIGHUP);
  }
  printf("%s terminating, pid: %d\n", prefix, pid);
  exit(0);
}
```

exit

A process terminates its execution with exit(retval).

exit(x) is called implicitly when main() terminates by the c runtime (wiki for crt.s)

the c runtime is a small stub of code that is linked in the executable during compilation and adds some glue.

a terminated process goes in the terminated (zombie) status, and all his resources are released.

A zombie process stays alive (as much as a zombie can be), until the parent reads its exit value via a wait/waitpid.

When this happens, the process ceases to exist.

Init/systemd periodically wait() for their children. This ensures that orphaned processes that terminate are not indefinitely zombies.

exec*

Replaces the memory map of an existing process with a new one, loaded from a program file.

When in "running" the process will start executing from _start (which is the routine in crto.s that calls main).

The memory of the process before calling exec* is dropped, together with all its resources.

Example of program that starts n instances of a program from command line.

```
int main(int argc, char** argv) {
  if (argc<3) {
    printf("usage %s <int> <path> <args>\n",
    argv[0]);
  }

  char* prog_path=argv[2];
  int num_instances=atoi(argv[1]);
  int active_instances=0;
  printf("starting program %s in %d instances\n",
         prog_path, num_instances);

  for (int i=0; i<num_instances; ++i){
    pid_t child_pid=fork();
    if(! child_pid){
      int result=execv(prog_path, argv+2);
      if (result) {
     printf("something wrong with exec errno=%s\n",
          strerror(errno));
      }
    } else
      active_instances++;
  }
  int status;
  while(active_instances) {
    pid_t child_pid = wait(&status);
    printf("son %d  died, mourning\n", child_pid);
    active_instances--;
  }
  printf(" launcher terminating\n");
  return 0;
}
```

exec and env

A process to run might require additional information

- environment variables
- main parameters (argc, argv)

These parameters can be passed to the exec* family of syscalls accepting arguments arguments

- exec
- execv (argv)
- execve (argv, environ)

the environment variables are accessible through a global NUL terminated string array

char** environ

each entry has the form

"NAME=VALUE"

last entry is 0

```
extern char** environ; // black magic here
int main(int argc, char** argv) {
  if (argc<3) {
     // banner
  }
  char* prog_path=argv[2];
  int num_instances=atoi(argv[1]);
  int active_instances=0;
  printf("starting program %s in %d instances\n",
          prog_path, num_instances);

  char* path=getenv("PATH");
  printf(" the current path is %s\n", path);

  for (int i=0; i<num_instances; ++i){
    pid_t child_pid=fork();
    if(! child_pid){
      int result=execvpe(prog_path, argv+2, environ);
      if (result) {
         printf("sth  wrong with exec errno=%s\n",
                strerror(errno));
      }
    } else
      active_instances++;

  }
  int status;
  while(active_instances) {
    pid_t child_pid = wait(&status);
    printf("son %d  died, morning\n", child_pid);
    active_instances--;
  }
  printf(" launcher terminating\n");
  return 0;
}
```

vfork

to start an executable, the only way with this schema is to

- create a new process
- exec

The creation of the new process involves useless operations

- duplicating file descriptors
- copying memory

If our aim is to do an exec immediately after forking (in the child), we can use vfork()

its behavior is the same as fork but it saves on copies

safe to use only if the first action after vfork in the child is exec

```
int main(int argc, char** argv) {
  if (argc<3) {
    printf("usage %s <int> <path> <args>\n",
    argv[0]);
  }

  char* prog_path=argv[2];
  int num_instances=atoi(argv[1]);
  int active_instances=0;
  printf("starting program %s in %d instances\n",
         prog_path, num_instances);

  for (int i=0; i<num_instances; ++i){
    pid_t child_pid=vfork();
    if(! child_pid){
      int result=execv(prog_path, argv+2);
      if (result) {
     printf("something wrong with exec errno=%s\n",
          strerror(errno));
      }
    } else
      active_instances++;
  }
  int status;
  while(active_instances) {
    pid_t child_pid = wait(&status);
    printf("son %d  died, mourning\n", child_pid);
    active_instances--;
  }
  printf(" launcher terminating\n");
  return 0;
}
```

Process Control Block

The kernel stores the information about a process in a data structure: the Process Control Block (PCB)

A typical PCB contains

- process ID (PID)
- user ID (UID)
- status of the program (ready, waiting...)
- CPU status for the process (registers)
- scheduling information*
- memory information (stack, page table*)
- I/O information (open descriptors*)

All in all from the PCB and the data structures linked by it one should be able to recover the state of a process.

The PCB is in a privileged memory area.

* on this screen, in the next episodes

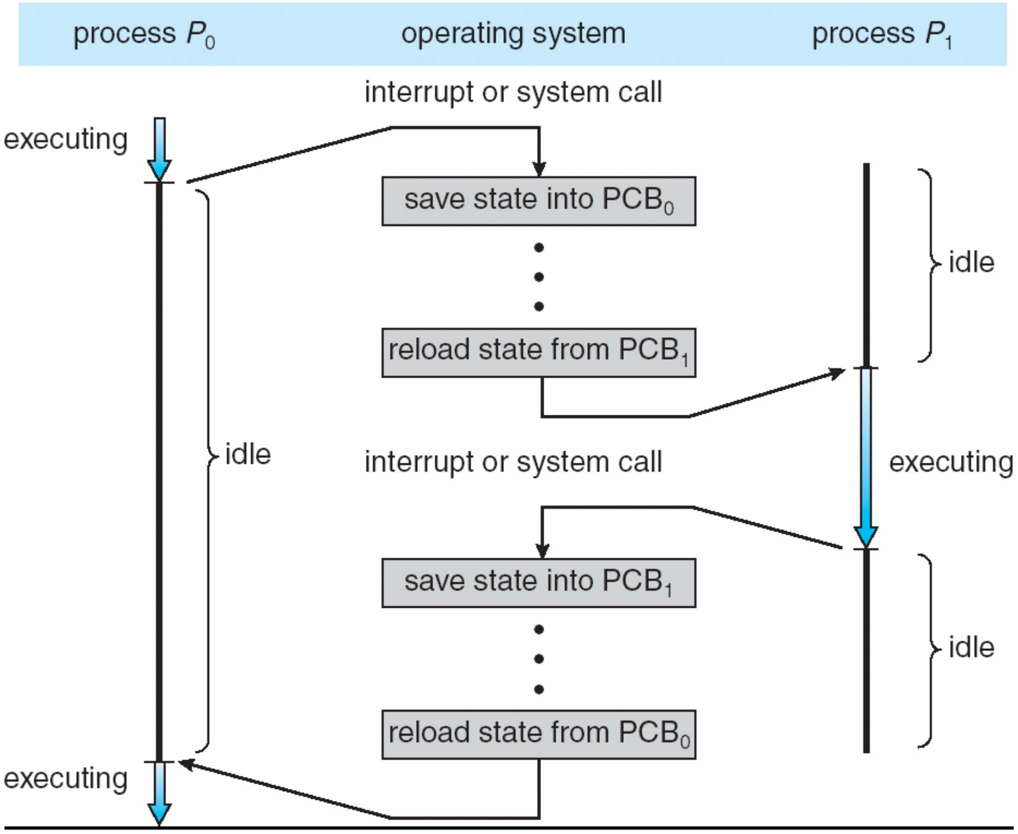
Context Switch

Occurs when a running process is interrupted.

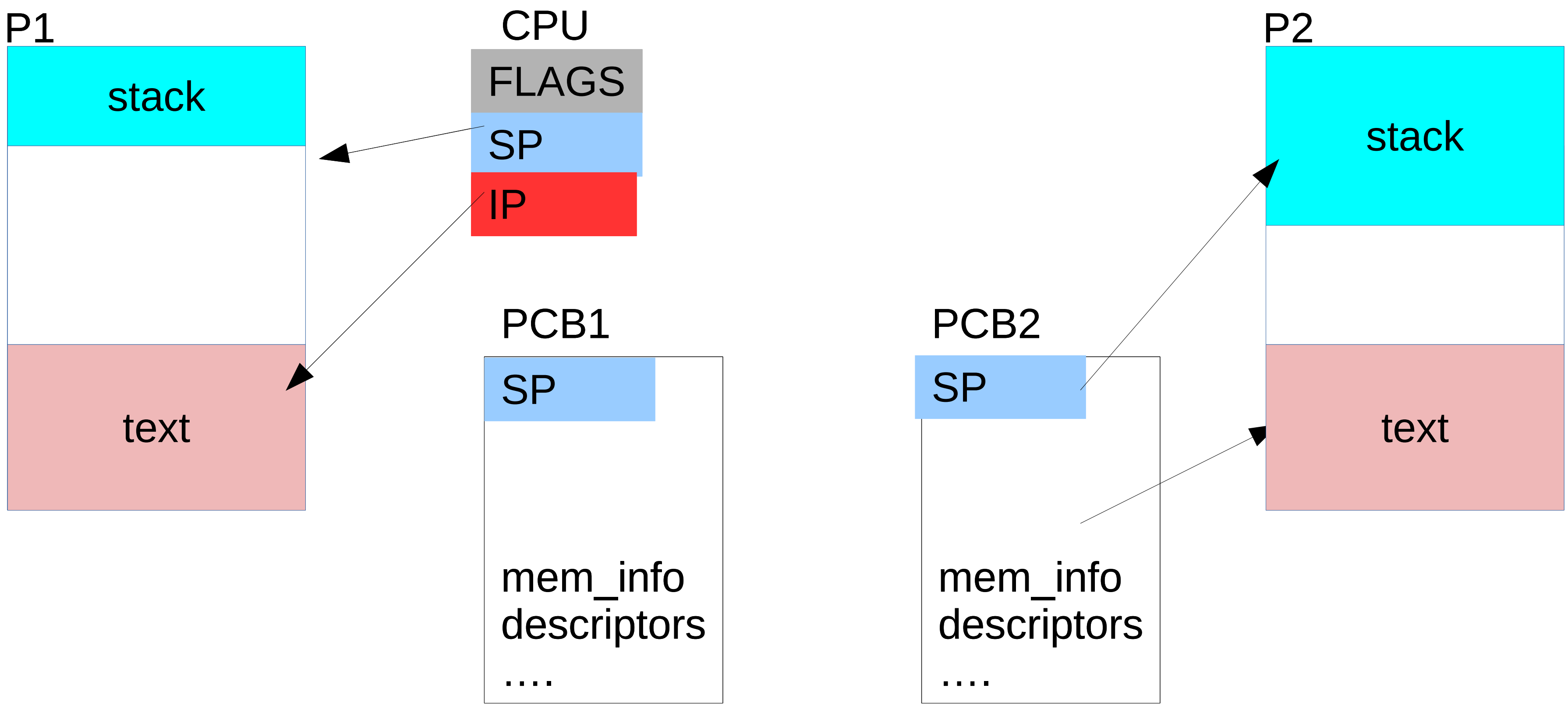
This can happen only because of an interrupt (or exception*).

These events cause the execution of kernel code.

When the kernel code returns, the next process that will enter the CPU might or might not be the one interrupted, depending on kernel’s decision.



Context Switch in Detail

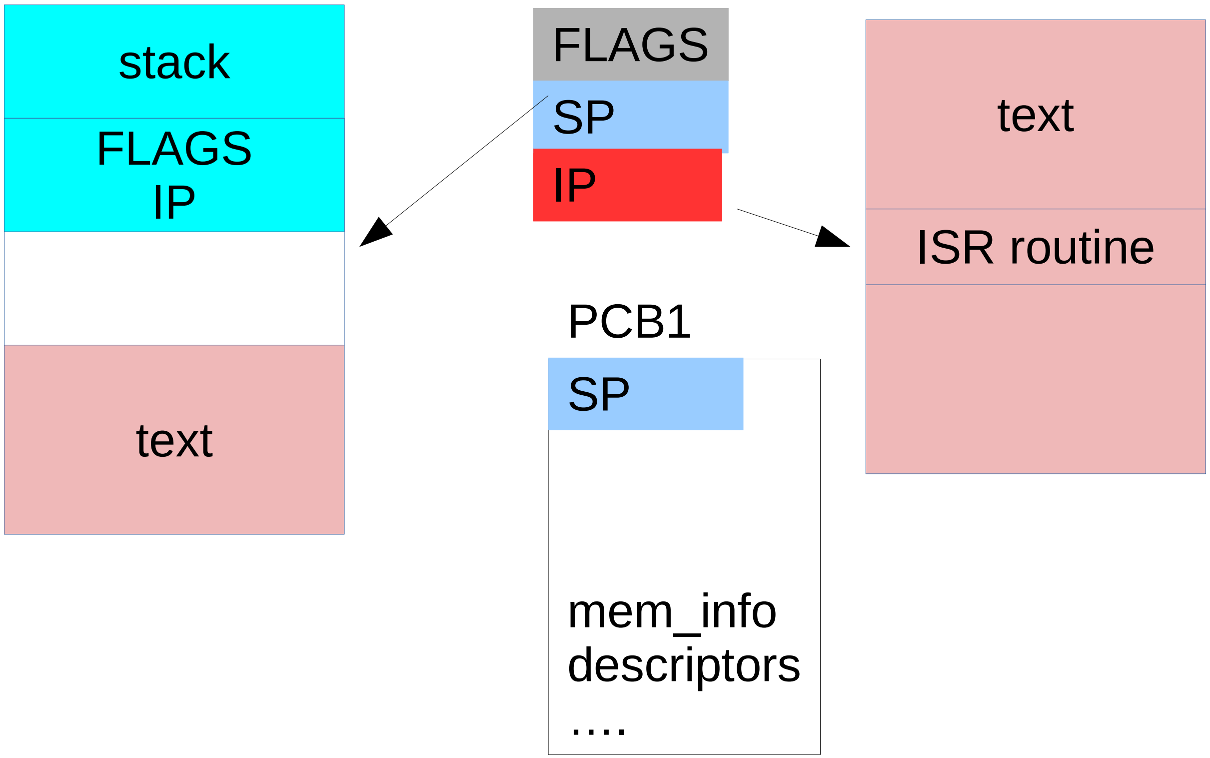


Scenario

- we have two processes in ready: P1 and P2
- P2 was previously sunning but it has been preempter. His status is in PCB2
- P1 is running

What should happen such that after an interrupt the CPU continues executing P2?

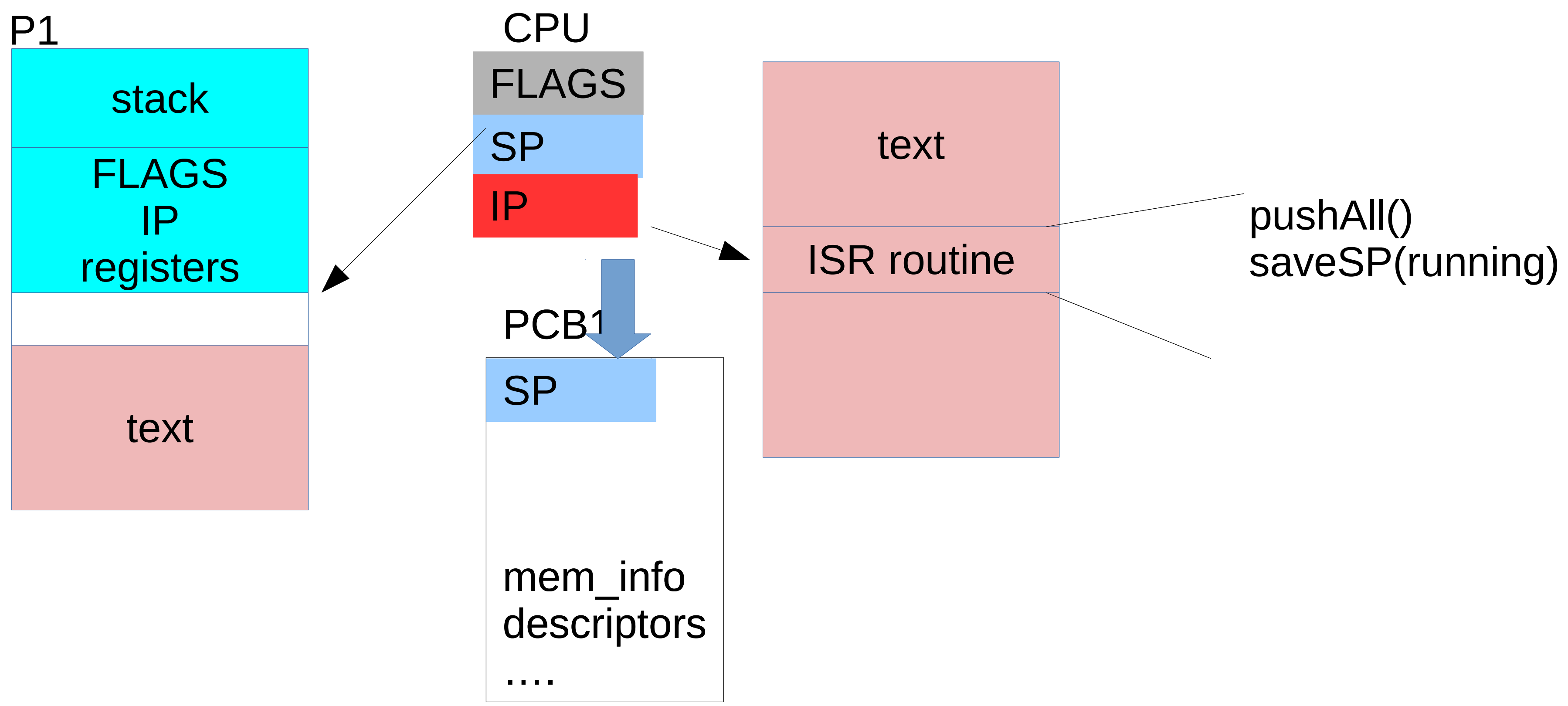
Context Switch in Detail



The interrupt comes, thus the CPU

- saves flags and instruction counter on the stack
- toggles to priviledged mode and handles the appropriate ISR

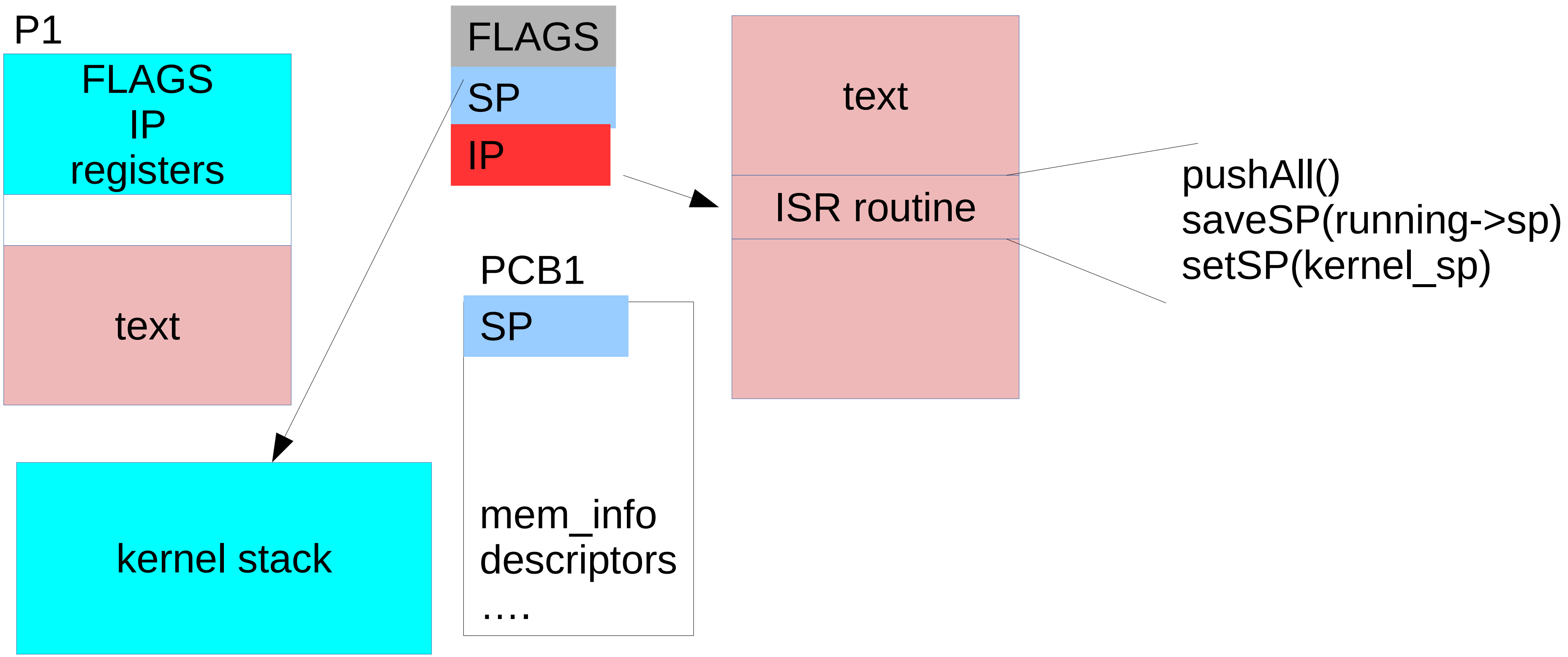
Context Switch in Detail



To recover P1 in the future, we need to save its CPU state in the PCB.

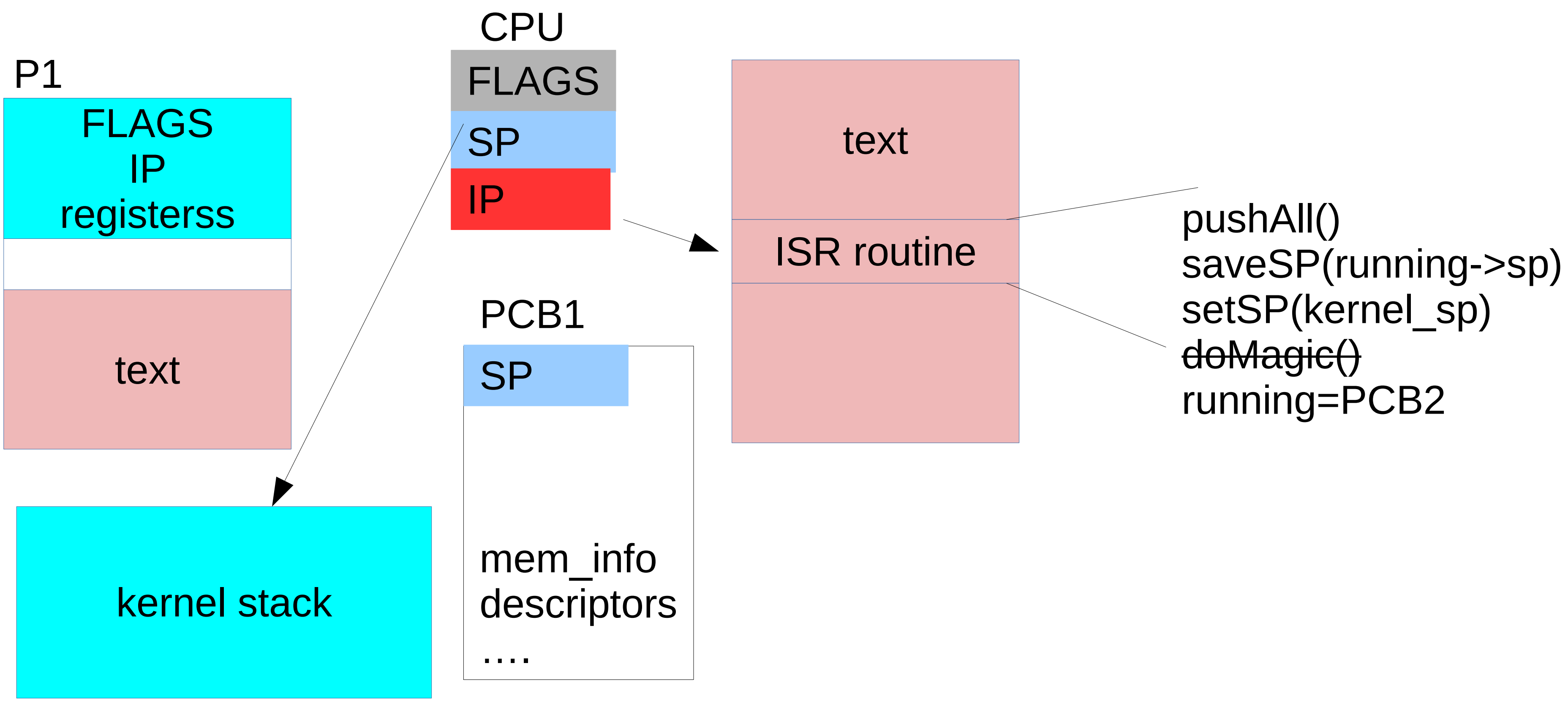
The state is on the stack, so in this example we save in the PCB just the stack pointer.

Context Switch in Detail



At this point we switch stack to the kernel stack. This step is optional, but it protects us from messing with the P1 stack.

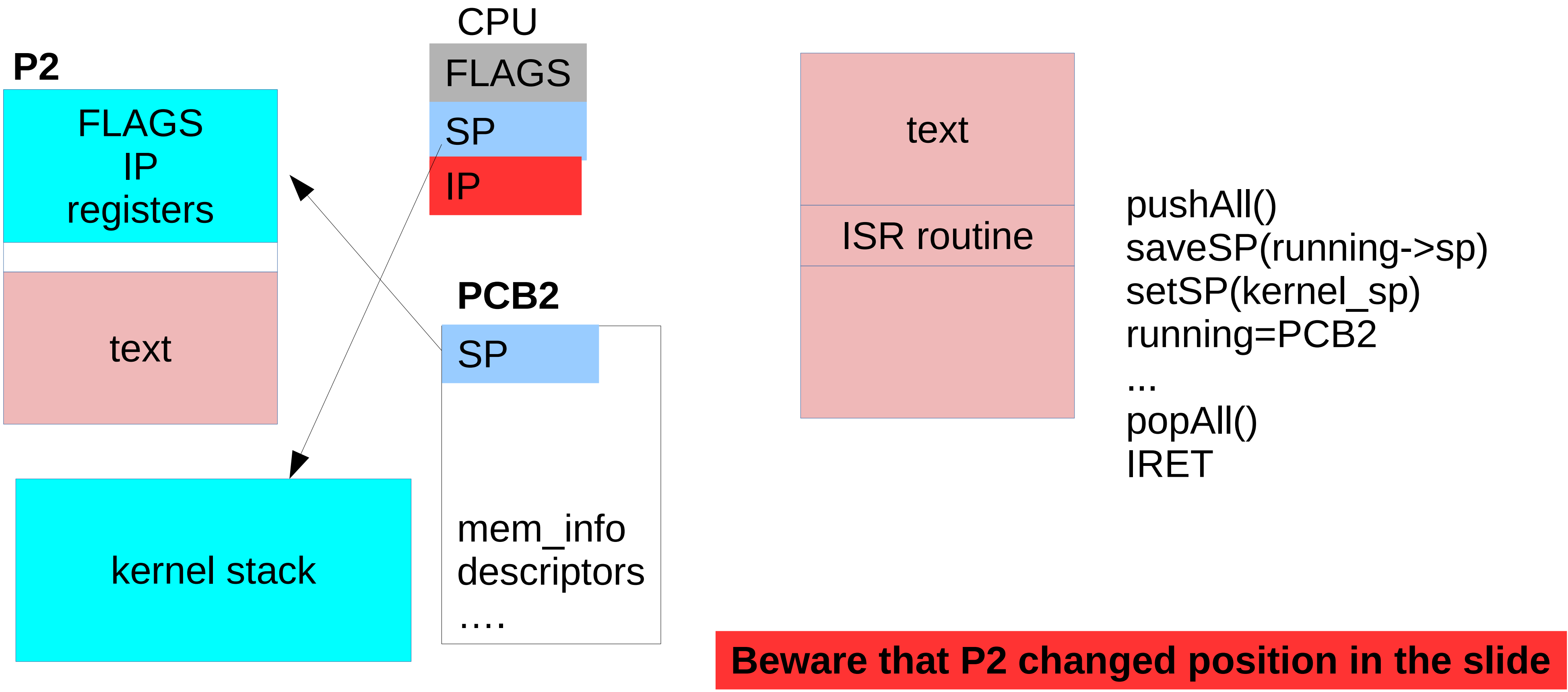
Context Switch in Detail



Once we change stack we do our task.

- If the trap was triggered by a syscall, we might want to look up for the parameters on the PCB or on P1's stack
- If our task is just to execute a context switch to P2, we need to select P2 as next running

Context Switch in Detail

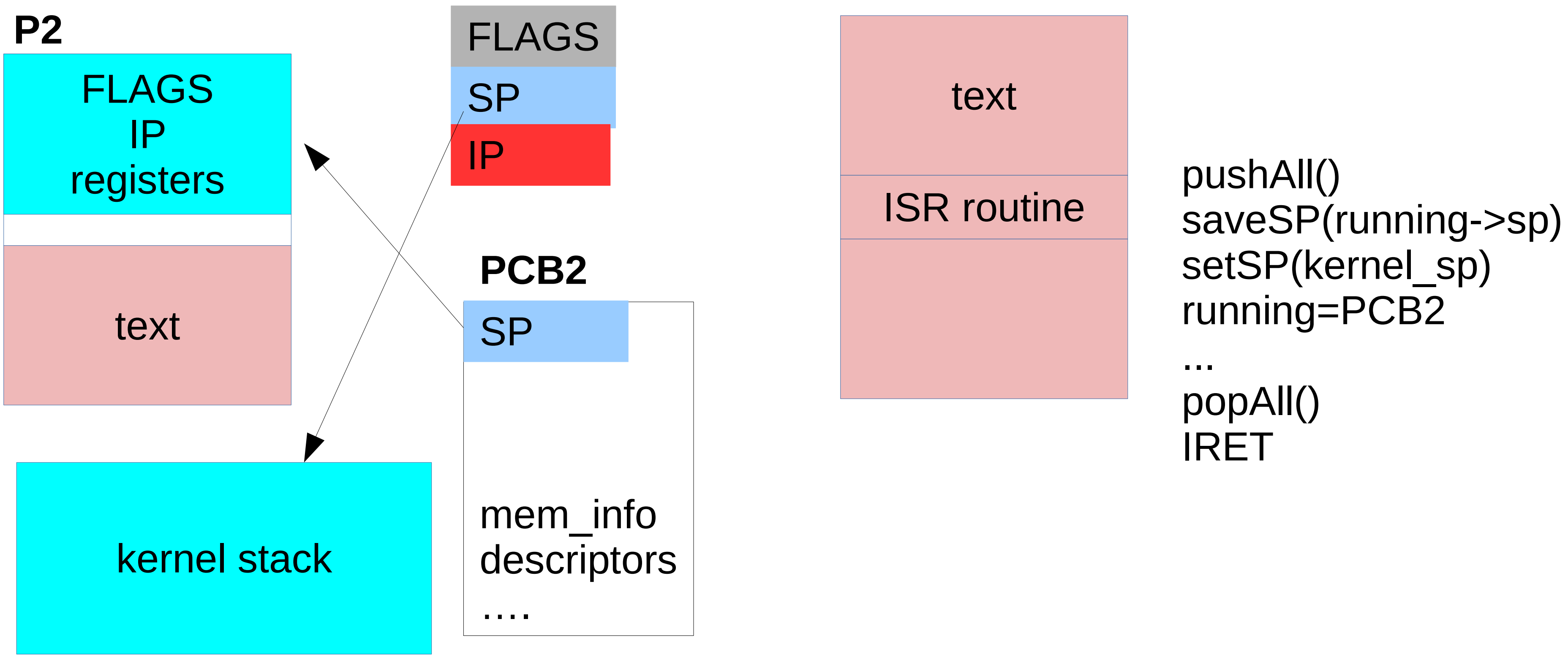


Let us assume P2 is our next running, we need to start it again.

Since P2 was preempted, we know its structures are consistent

We know that the last instruction being executed in kernel mode will be a return from interrupt (IRET), that recovers the flags.

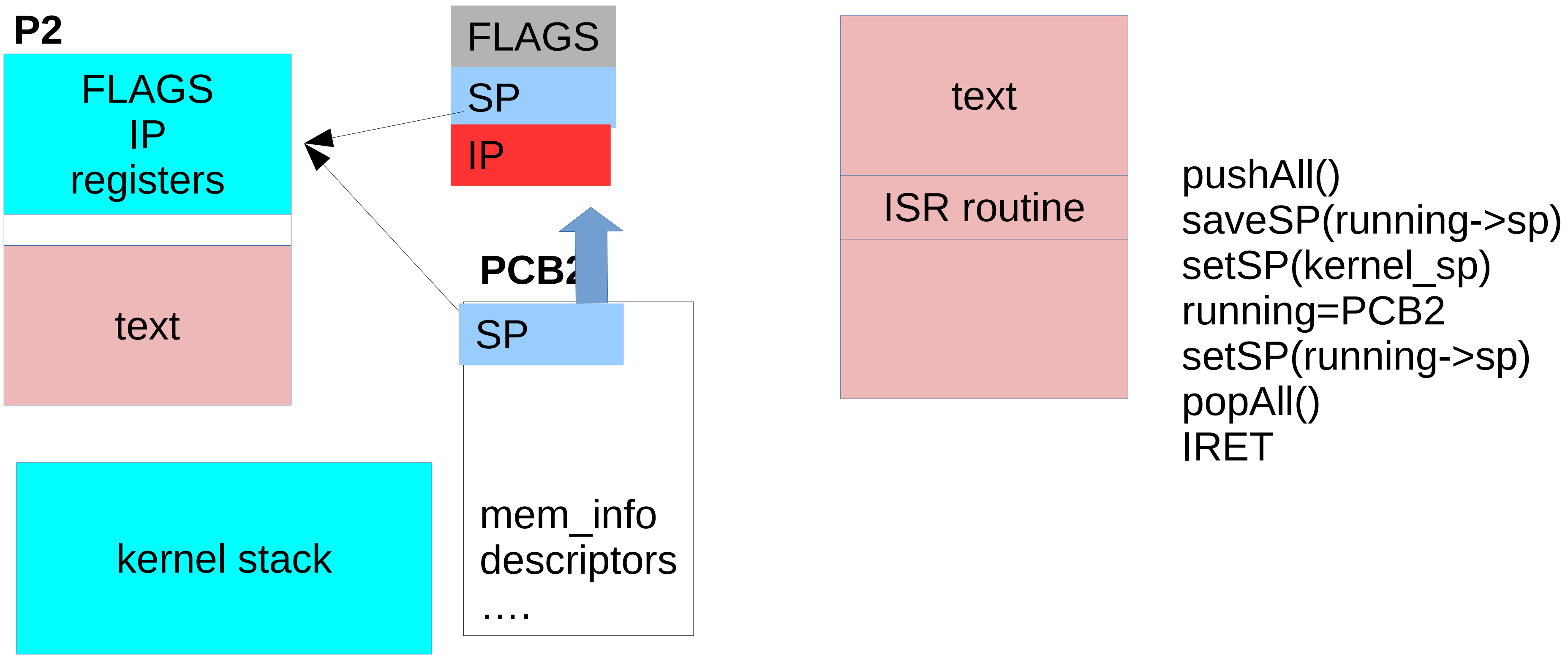
Context Switch in Detail



For the IRET to work, we need to assume the stack consistent.

This is verified since P2 was assumed to be preempted thus we "left" the stack untouched, after saving the registers

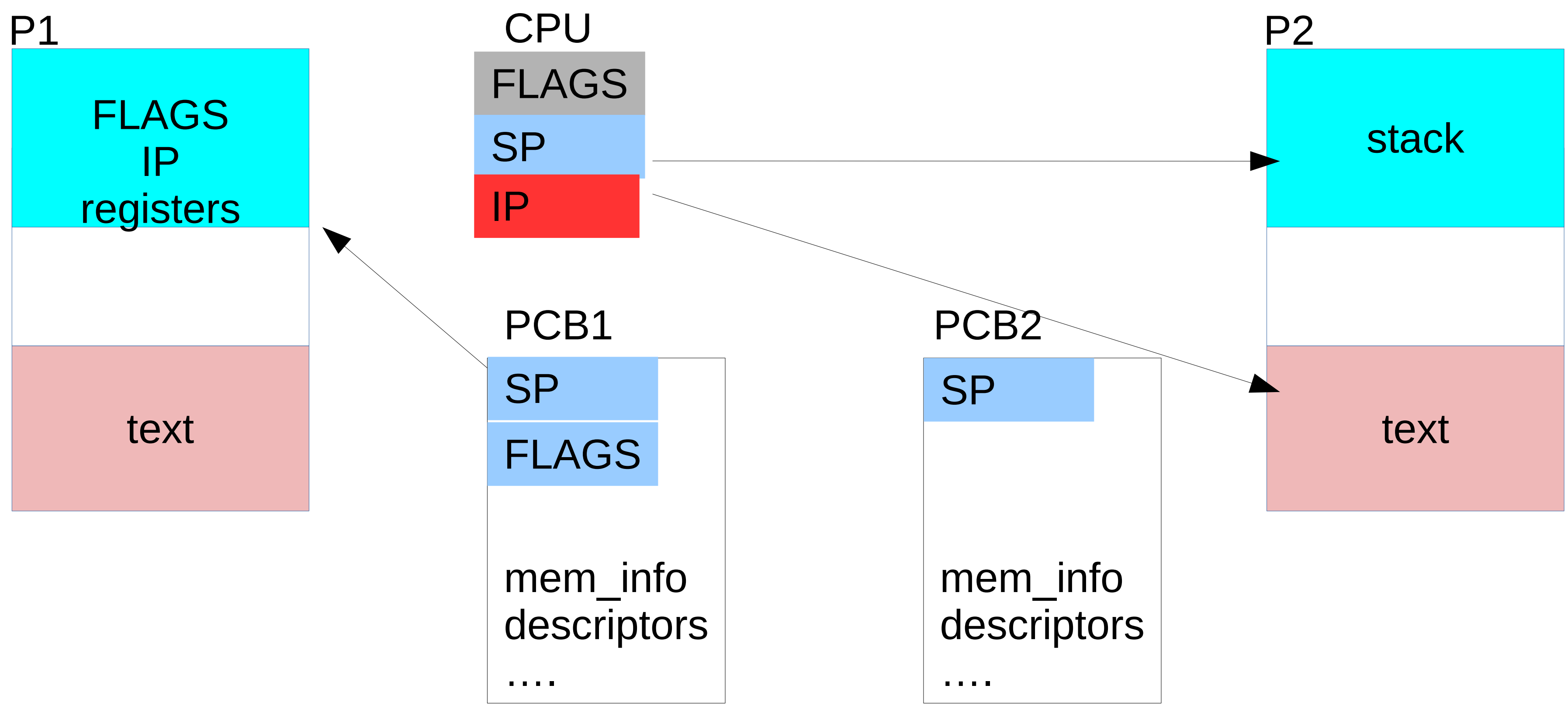
Context Switch in Detail



To continue the execution, we

- change the stack back resding it from the running pcb
- restore the state in the CPU
- return from interrupt

Context Switch in Detail



Et voila' P2 is running again as if nothing has happened

Preamble and Postamble

```
pushAll()
saveSP(running->sp)
setSP(kernel_sp)
```

```
doMagic()
```

```
setSP(running->sp)
popAll()
IRET
```

A generic ISR does not follow usual C calling conventions.

- The entry/exit in kernel mode is has a preamble and postamble, and have the role of ensuring a proper restoring of the process, and interaction with the kernel structures.
- Assembly needed for manipulating registers (SP, push).
- If a syscall wants to read some argument, it retrieves them from the stack of the current process (or from the registers), accessible through the SP saved in the current pcb.
- returning values done by altering the stack of the current PCB, in the area of the saved registers.

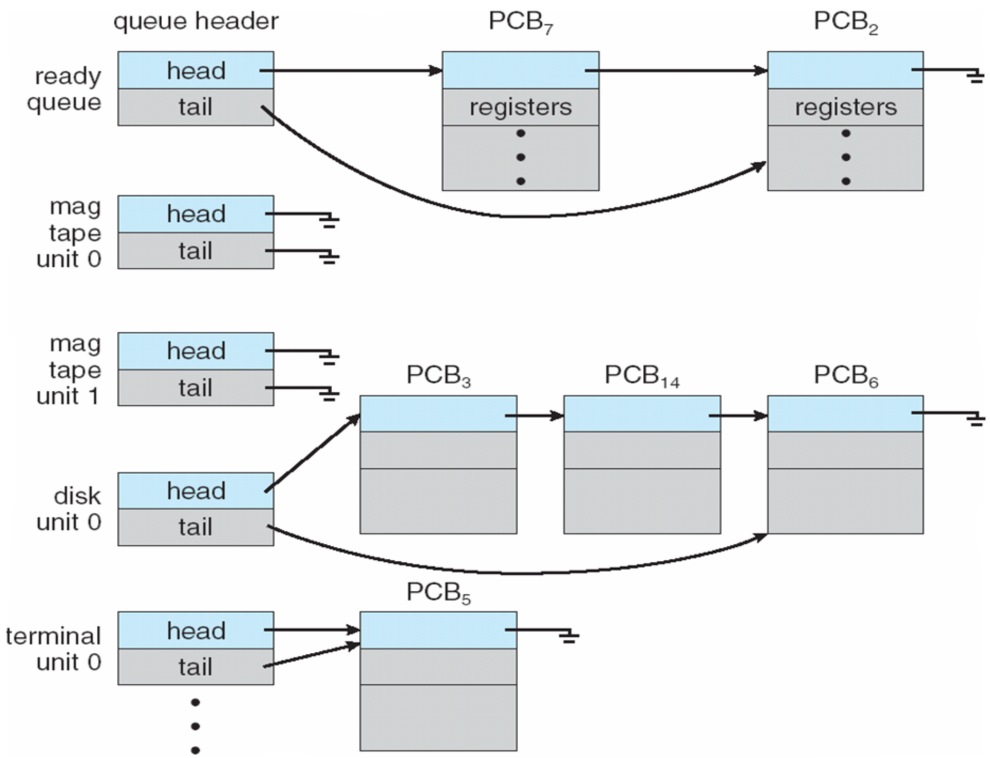
Process Scheduler

Chooses the next process that gets the CPU between the processes being executed.

Its characteristics depends on the application/context

- mainframe: maximize usage of resources
- desktop: minimize reaction times

The scheduler uses queues to handle process in execution, usually implemented as linked lists of PCBs



Scheduler Schema