

CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu



Lecture 3: Processes
February 8, 2007

Assignment 0

Assignment #0 has been released on the course website

Goal: Familiarize you with the OS/161 system

- Set up your CVS tree
- Compile and run a basic OS/161 kernel
- Some simple code-reading questions

Due: Tuesday, Feb 13, in class

CS161 Bulletin Board system set up – see course website

Process Management

This lecture begins a series of topics on

- Processes
- Threads
- Synchronization

Probably the most important part of the class!

- Definitely will be questions on this on the midterm and final
- Important to get this stuff down to do Assignments 1 and 2

Today: Processes and process management

- What are the units of execution inside of an OS?
- How are they implemented inside of the OS?
- What are the possible execution states of a process?
- How does the OS switch between processes?

What is a process?

A *process* is the OS's abstraction for *execution*

- A process represents a single running application on the system

Process has three main components:

1) Address space

- The memory that the process can access
- Consists of various pieces: the program code, static variables, heap, stack, etc.

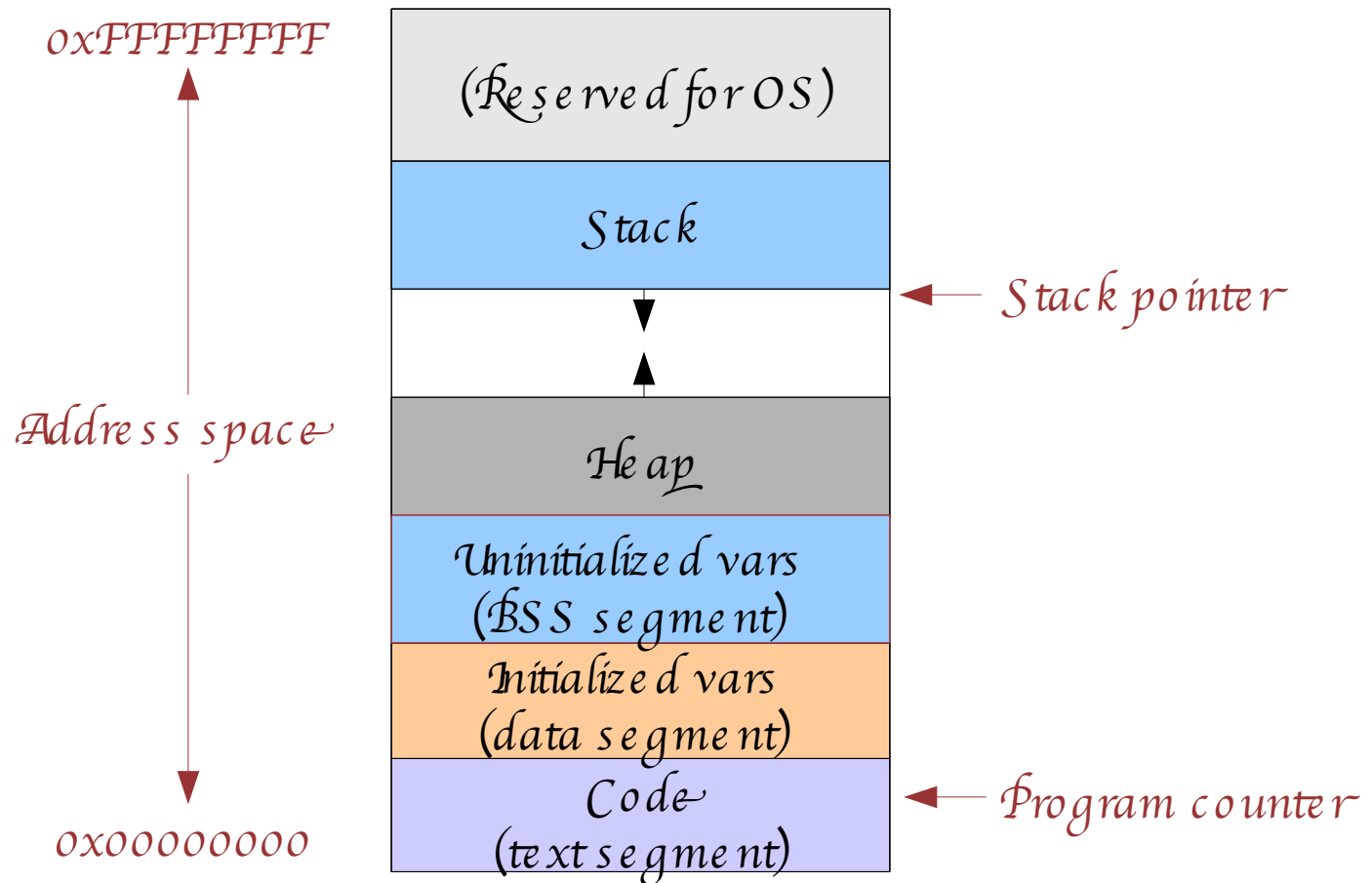
2) Processor state

- The CPU registers associated with the running process
- Includes general purpose registers, program counter, stack pointer, etc.

3) OS resources

- Various OS state associated with the process
- Examples: open files, network sockets, etc.

Process address space



- The range of **virtual memory addresses** that the process can access
- Includes the code of the running program
- The data of the running program (static variables and heap)
- An execution stack
 - *Local variables and saved registers for each procedure call*

Mysteries of the address space

The address space is how the process sees its own memory

- Not necessarily how the memory is laid out in physical RAM.

Every process has its own **separate** address space.

- Memory addresses are “private” to each process.
- If Process A accesses address 0x3000, and Process B accesses 0x3000, they are accessing **different** physical memory locations.
 - *Unless, of course, the processes have been set up to share memory somehow.*

On a 32-bit machine, each process can address 2^{32} bytes (4GB) of memory!

- How is this possible if you have less than 4GB on your machine?

Mysteries of the address space

The address space is how the process sees its own memory

- Not necessarily how the memory is laid out in physical RAM.

Every process has its own **separate** address space.

- Memory addresses are “private” to each process.
- If Process A accesses address 0x3000, and Process B accesses 0x3000, they are accessing **different** physical memory locations.
 - *Unless, of course, the processes have been set up to share memory somehow.*

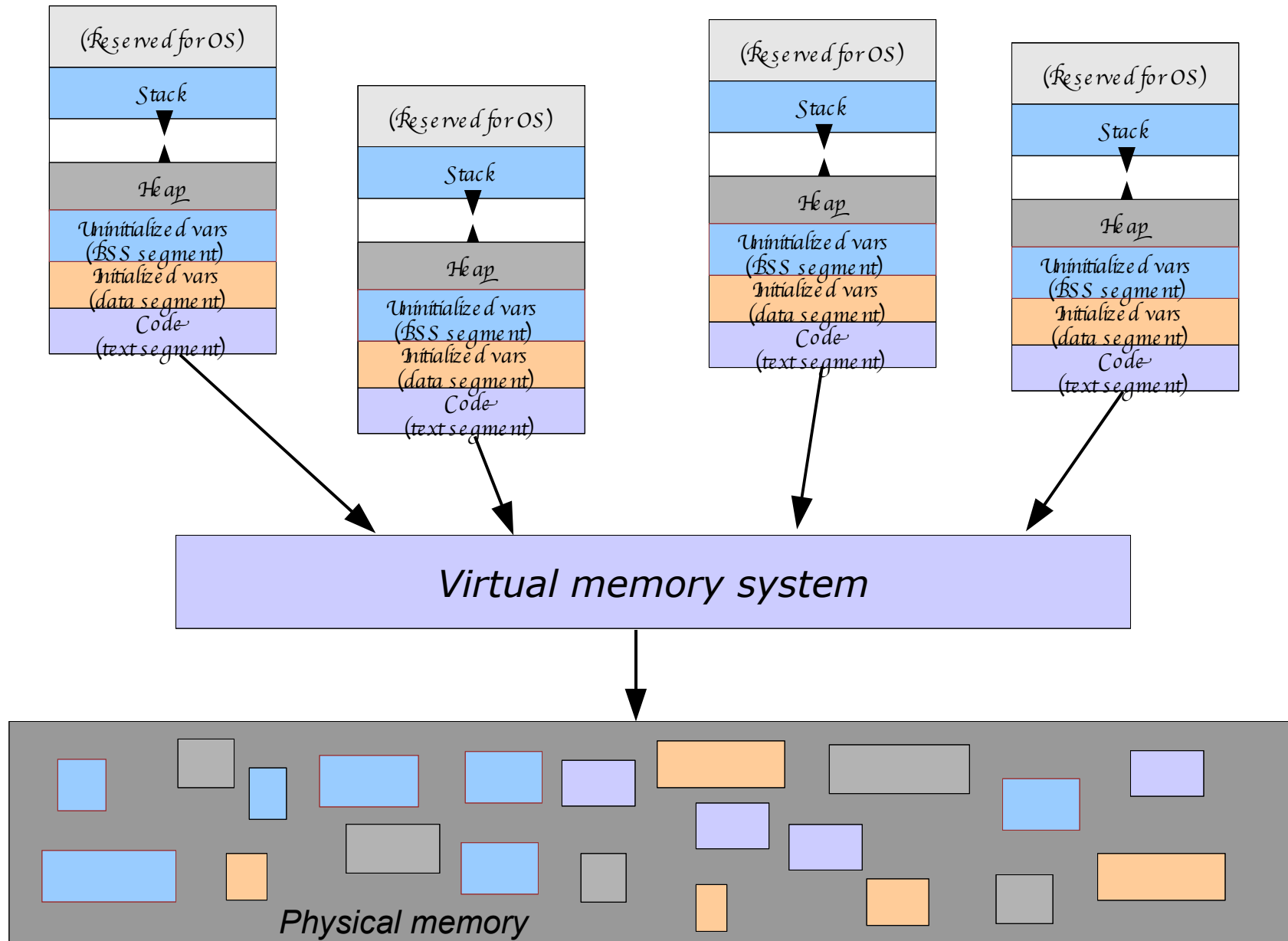
On a 32-bit machine, each process can address 2^{32} bytes (4GB) of memory!

- How is this possible if you have less than 4GB on your machine?

This elaborate illusion is provided by the **virtual memory system**

- We will discuss all of this in painful detail later in the class.

What happens with multiple processes?



Process States

Each process has an *execution state*

- Indicates what the process is currently doing

Running:

- Process is currently using the CPU

Ready:

- Currently waiting to be assigned to a CPU
- That is, the process could be running, but another process is using the CPU

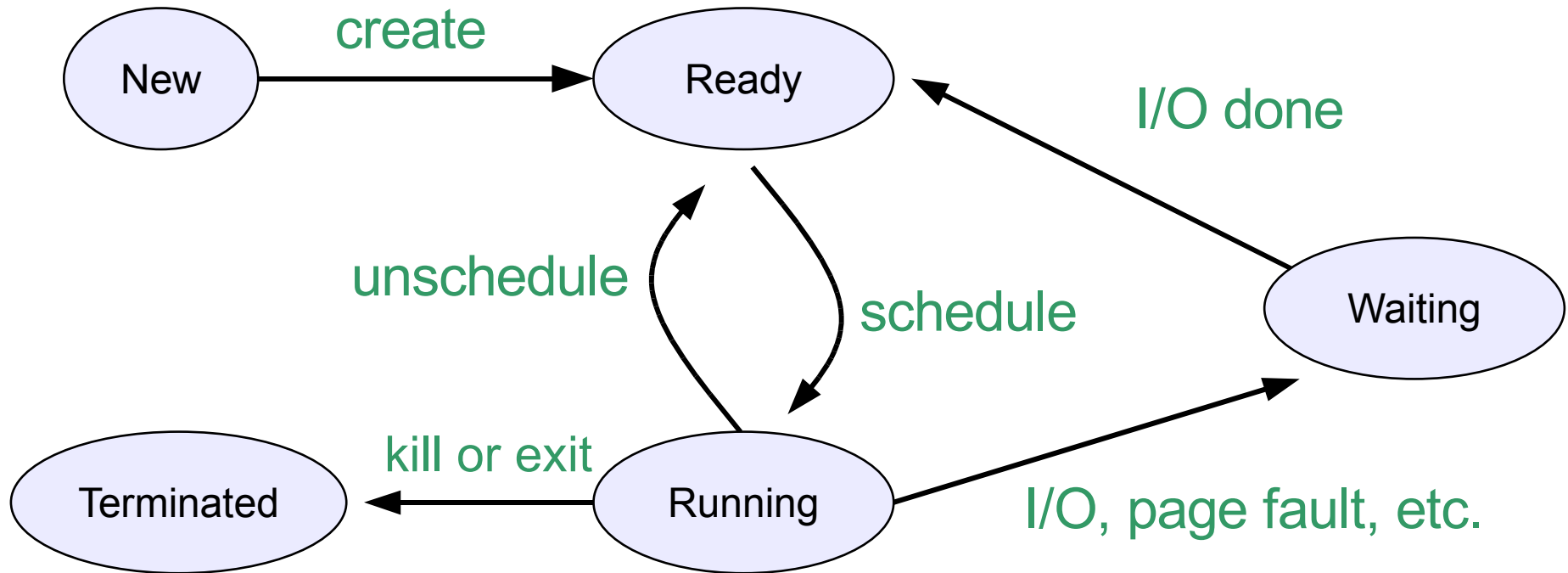
Waiting (or sleeping or blocked):

- Process is waiting for an event
- Such as completion of an I/O, a timer to go off, etc.
- Why is this different than “ready” ?

As the process executes, it moves between these states

- What state is the process in most of the time?

Process State Transitions



- What causes schedule and unschedule transitions?

Process Control Block

OS maintains a *Process Control Block (PCB)* for each process

The PCB is a big data structure with many fields:

- Process ID
- User ID
- Execution state
 - *ready, running, or waiting*
- Saved CPU state
 - *CPU registers saved the last time the process was suspended.*
- OS resources
 - *Open files, network sockets, etc.*
- Memory management info
- Scheduling priority
 - *Give some processes higher priority than others*
- Accounting information
 - *Total CPU time, memory usage, etc.*

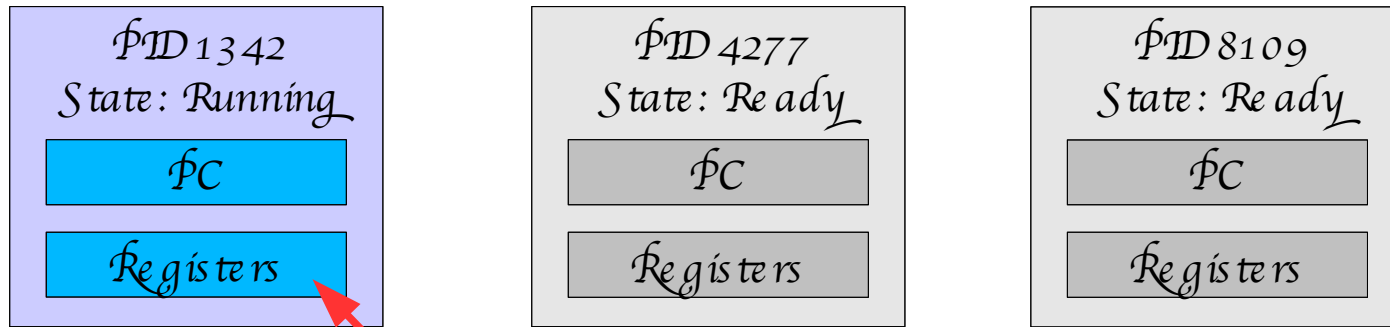
Linux PCB Structure (task_struct)

```
struct task_struct {
volatile long state; Execution state
unsigned long flags;
int sigpending;
mm_segment_t addr_limit;
struct exec_domain *exec_domain;
volatile long need_resched;
unsigned long ptrace;
int lock_depth;
unsigned int cpu;
int prio, static_prio;
struct list_head run_list;
prio_array_t *array;
unsigned long sleep_avg;
unsigned long last_run;
unsigned long policy;
unsigned long cpus_allowed;
unsigned int time_slice, first_time_slice;
atomic_t usage;
struct list_head tasks;
struct list_head ptrace_children;
struct list_head ptrace_list;
struct mm_struct *mm, *active_mm; Memory mgmt info
struct linux_binfmt *binfmt;
int exit_code, exit_signal;
int pdeath_signal;
unsigned long personality;
int did_exec:1;
unsigned task_dumpable:1;
pid_t pid; Process ID
pid_t pgrp;
pid_t tty_old_pgrp;
pid_t session;
pid_t tgid;
int leader;
struct task_struct *real_parent;
struct task_struct *parent;
struct list_head children;
struct list_head sibling;
struct task_struct *group_leader;
struct pid_link pids[PIDTYPE_MAX];
wait_queue_head_t wait_chldexit;
struct completion *vfork_done;
int *set_child_tid;
int *clear_child_tid;
unsigned long rt_priority; Priority
```

```
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
struct tms times;
struct tms group_times;
unsigned long start_time;
long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt,
cnsnap;
int swappable:1;
uid_t uid, euid, suid, fsuid; User ID
gid_t gid, egid, sgid, fsgid;
int ngroups;
gid_t groups[NGROUPS];
kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
int keep_capabilities:1;
struct user_struct *user;
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
int link_count, total_link_count;
struct tty_struct *tty;
unsigned int locks;
struct sem_undo *semundo;
struct sem_queue *semsleeping;
struct thread_struct thread; CPU state
struct fs_struct *fs;
struct files_struct *files; Open files
struct namespace *namespace;
struct signal_struct *signal;
struct sighand_struct *sighand;
sigset_t blocked, real_blocked;
struct sigpending pending;
unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;
void *tux_info;
void (*tux_exit)(void);
u32 parent_exec_id;
u32 self_exec_id;
spinlock_t alloc_lock;
spinlock_t switch_lock;
void *journal_info;
unsigned long ptrace_message;
siginfo_t *last_siginfo;
};
```

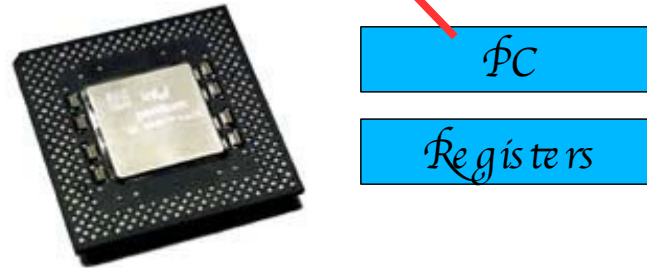
Context Switching

- The act of swapping a process state on or off the CPU is a *context switch*



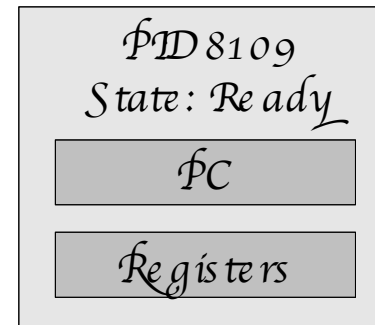
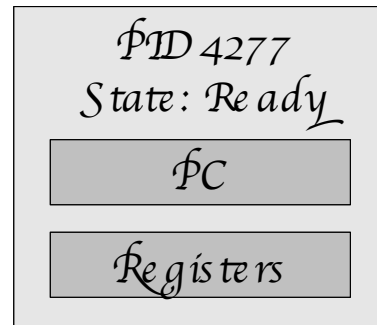
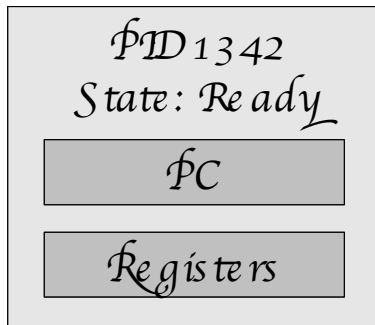
Currently running process

Save current CPU state



Context Switching

- The act of swapping a process state on or off the CPU is a *context switch*

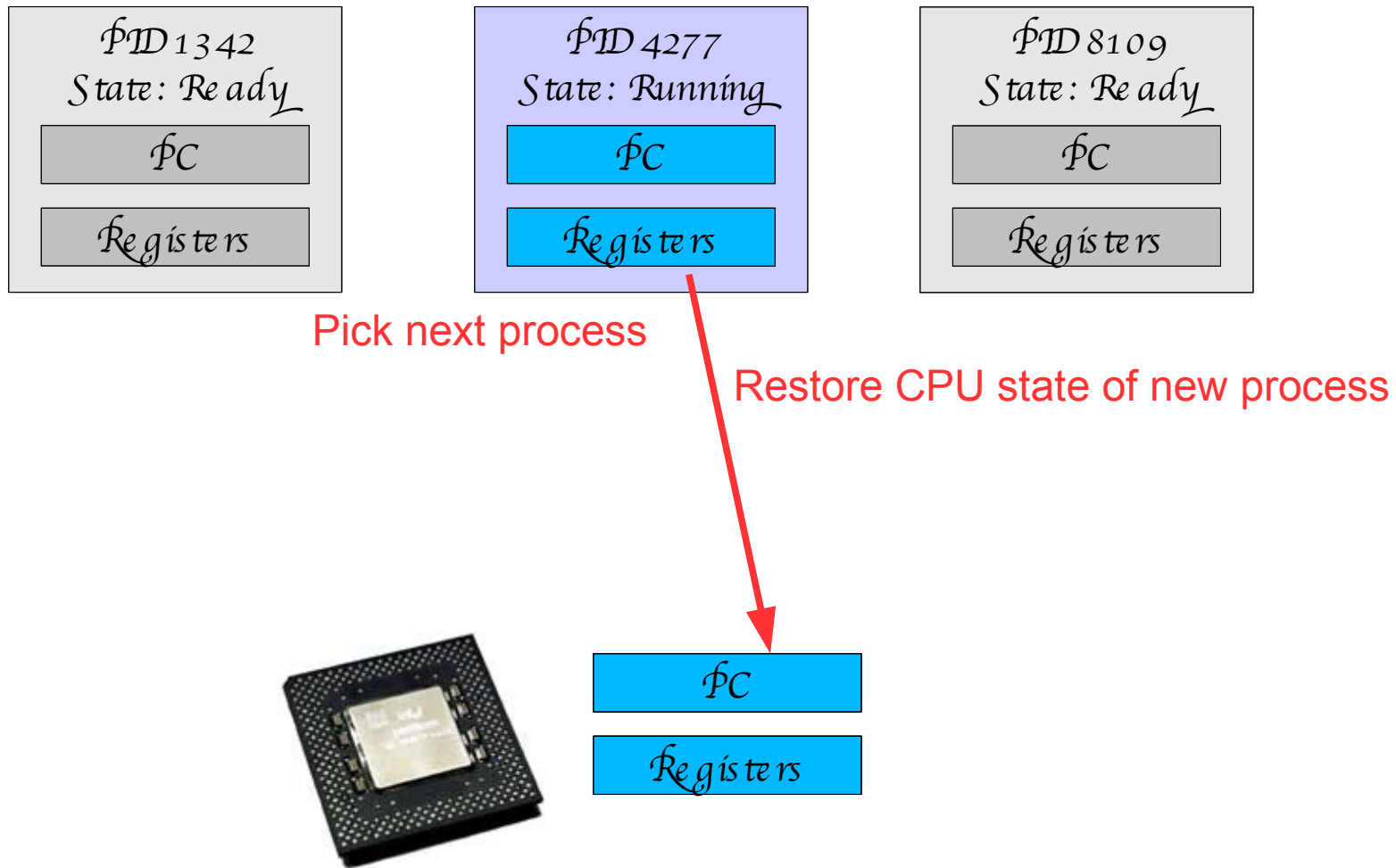


Suspend process



Context Switching

- The act of swapping a process state on or off the CPU is a *context switch*



Context Switch Overhead

Context switches are not cheap

- Generally have a lot of CPU state to save and restore
- Also must update various flags in the PCB
- Picking the next process to run – *scheduling* – is also expensive

Context switch overhead in Linux 2.4.21

- About 5.4 usec on a 2.4 GHz Pentium 4
- This is equivalent to about 13,200 CPU cycles!
 - *Not quite that many instructions since CPI > 1*

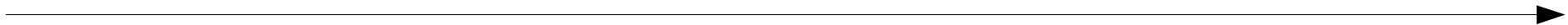
Context Switching in Linux



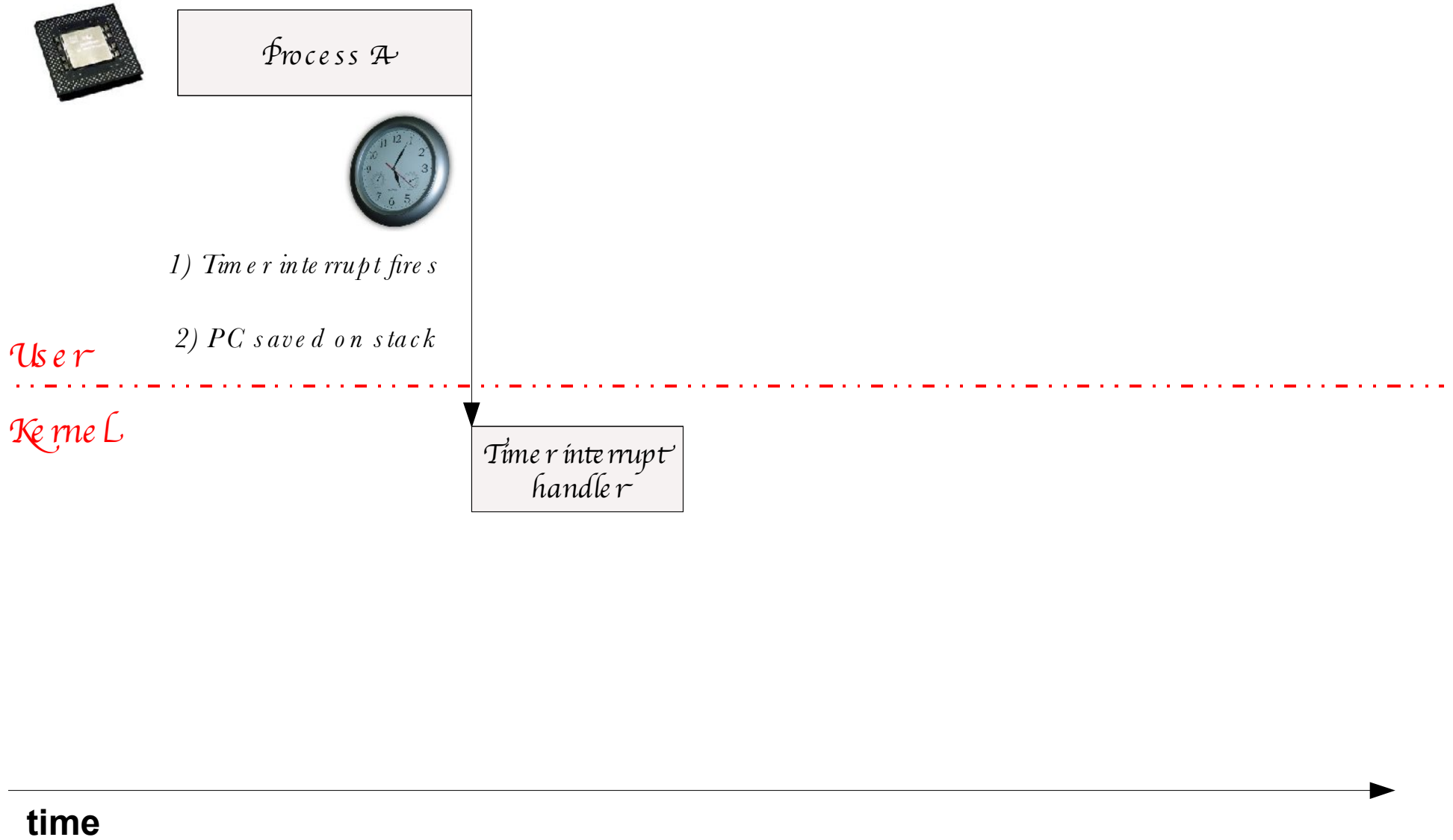
Process A

Process A is happily running along...

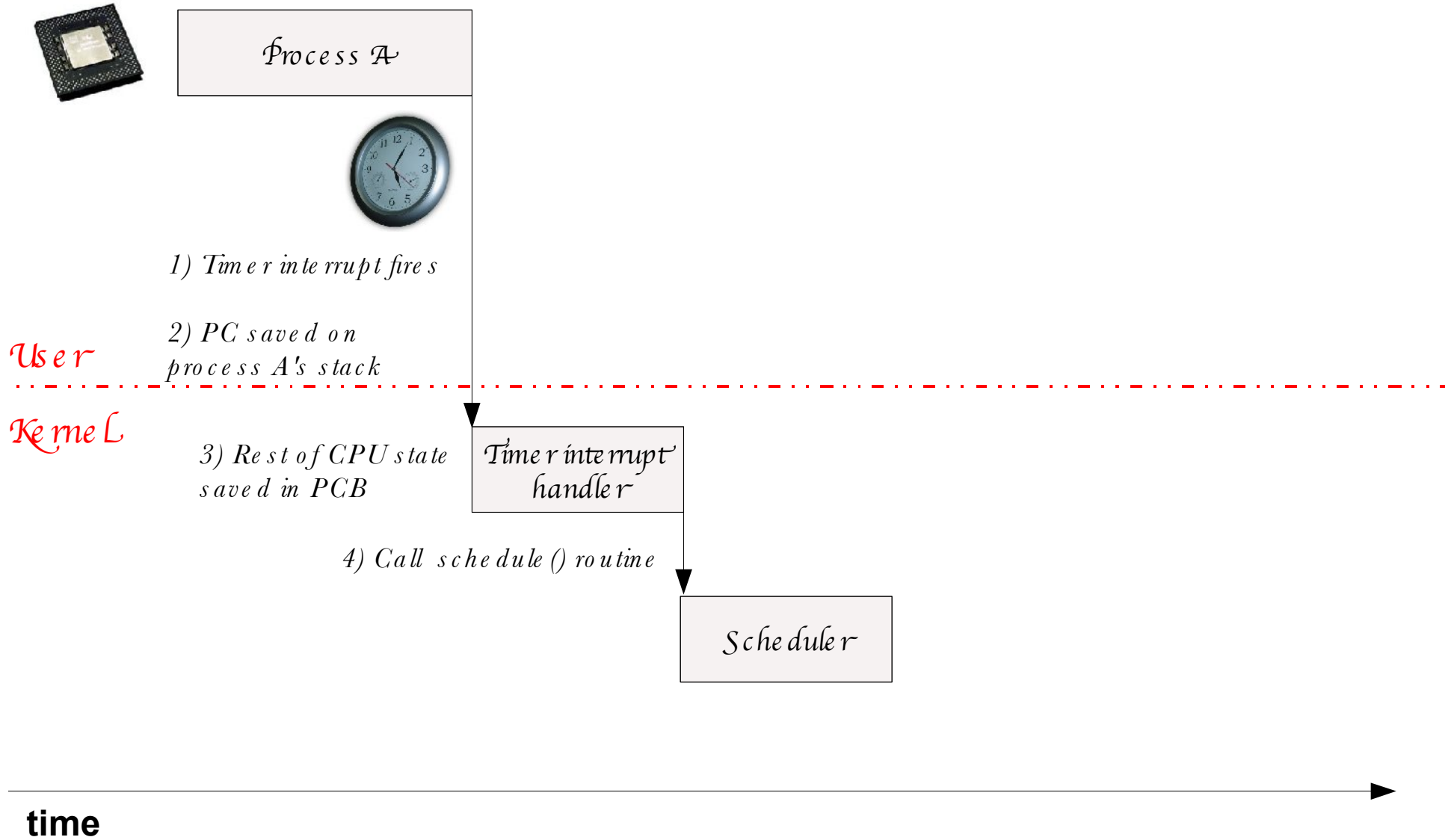
time



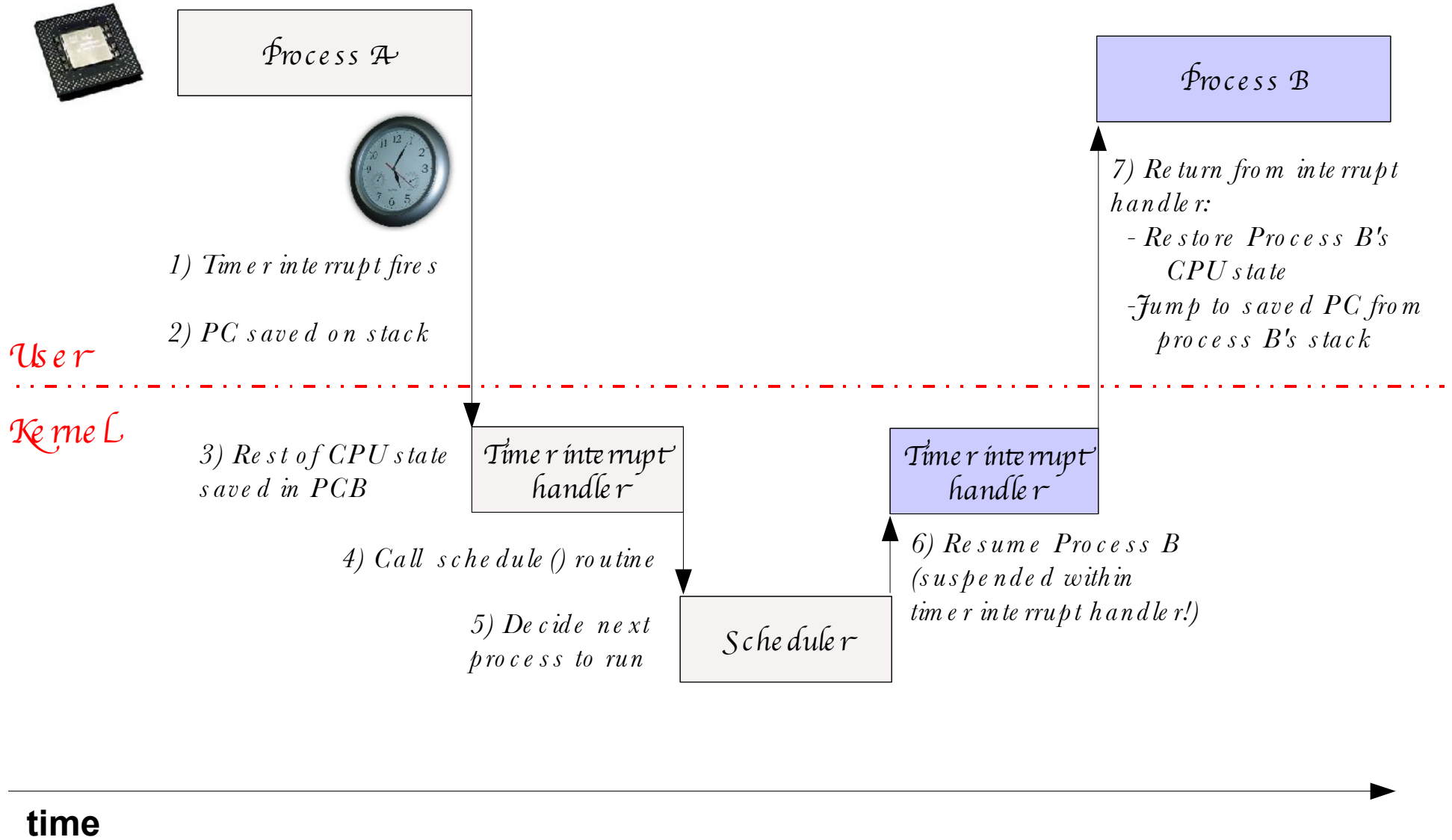
Context Switching in Linux



Context Switching in Linux



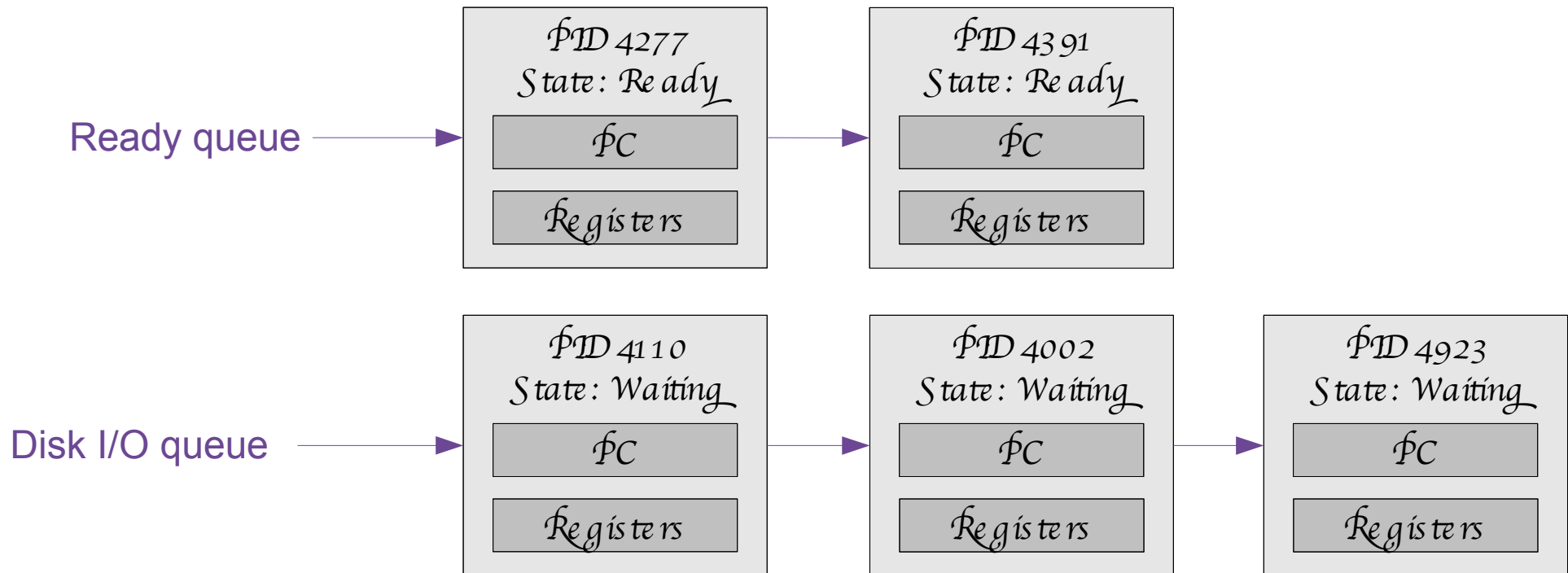
Context Switching in Linux



State Queues

The OS maintains a set of *state queues* for each process state

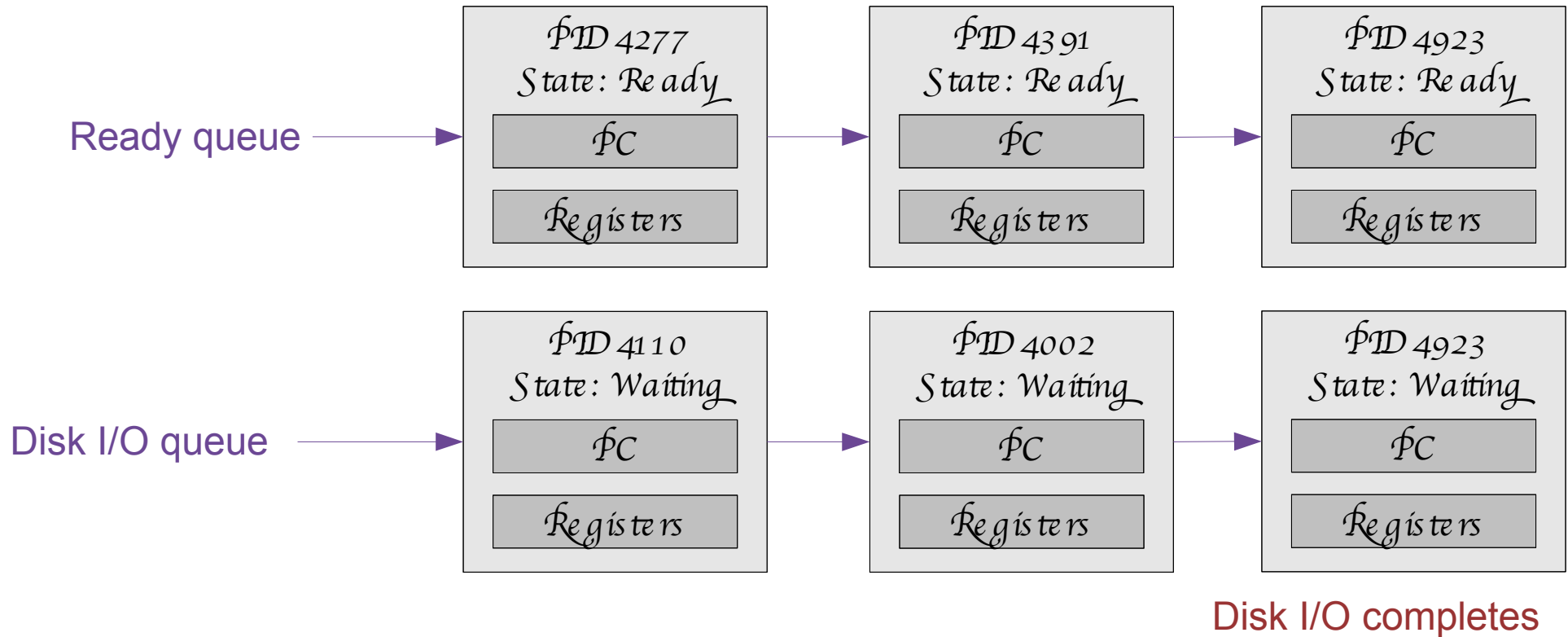
- Separate queues for ready and waiting states
- Generally separate queues for each kind of waiting process
 - e.g., One queue for processes waiting for disk I/O
 - Another queue for processes waiting for network I/O, etc.



State Queue Transitions

PCBs move between these queues as their state changes

- When scheduling a process, pop the head off of the ready queue
- When I/O has completed, move PCB from waiting queue to ready queue



Process Creation

One process can create, or *fork*, another process

- The original process is the *parent*
- The new process is the *child*

```
% pstree -p
```

```
init(1)-+-apmd(687)
          |
          | -atd(847)
          | -crond(793)
          | -rxvt(2700)---bash(2702)---ooffice(2853)
          | -rxvt(2752)---bash(2754)
```

- What creates the first process in the system, and when?

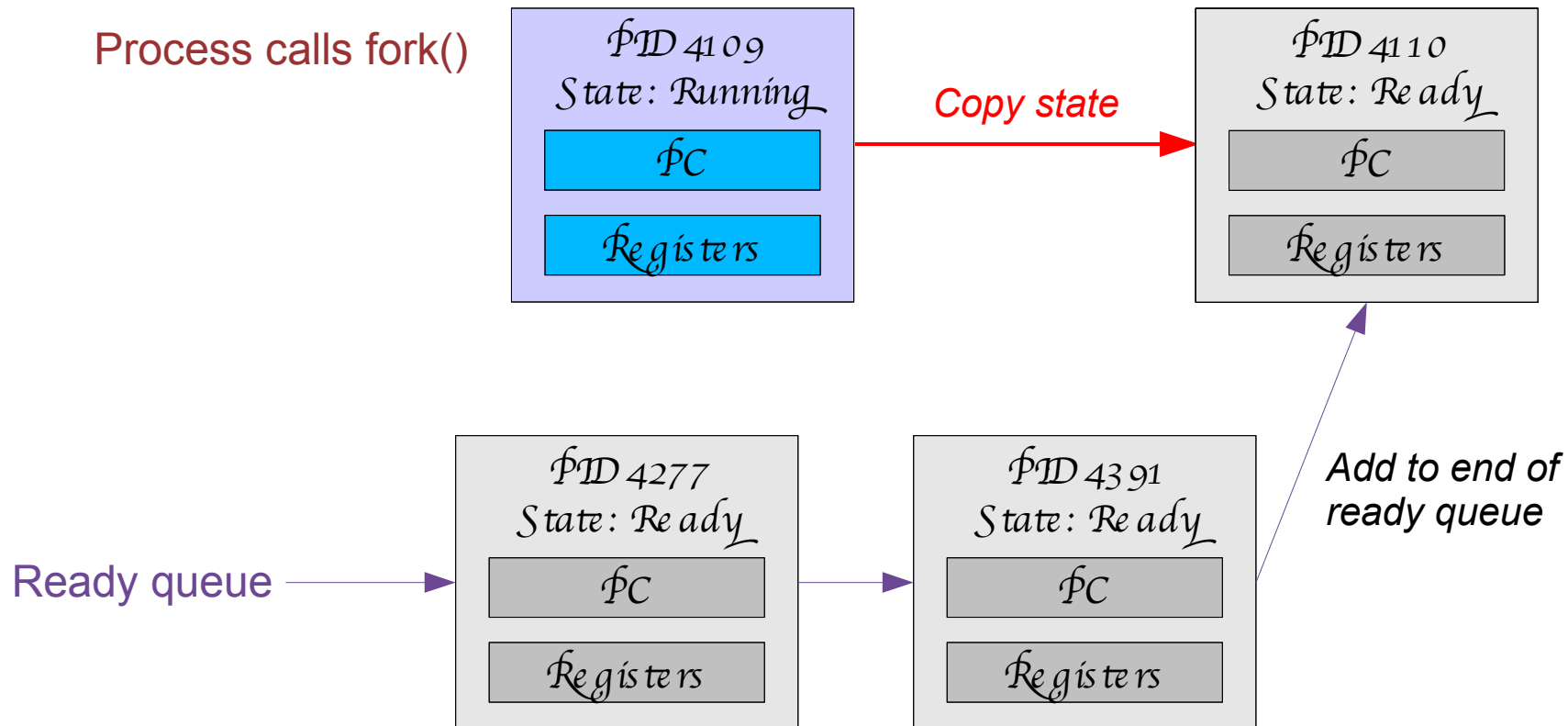
Parent process defines resources and access rights of children

- Just like real life ...
- e.g., child process inherits parent's user ID

UNIX fork mechanism

In UNIX, use the `fork()` system call to create a new process

- This creates an **exact duplicate** of the parent process!!
- Creates and initializes a new PCB
- Creates a new address space
- Copies entire contents of parent's address space into the child
- Initializes CPU and OS resources to a copy of the parent's
- Places new PCB on ready queue



UNIX fork mechanism

New child process starts running where `fork()` system call returns!

- And has an exact copy of the parent's variables.

```
int main(int argc, char **argv) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("Process %d: value is %d\n", getpid(), i);

        if (i == 5) {
            printf("Process %d: About to do a fork...\n", getpid());
            int child_pid = fork();
        }
    }
}
```

Output of sample program

```
Process 4530: value is 0
Process 4530: value is 1
Process 4530: value is 2
Process 4530: value is 3
Process 4530: value is 4
Process 4530: value is 5
Process 4530: About to do a fork...
Process 4531: value is 6
Process 4530: value is 6
Process 4530: value is 7
Process 4531: value is 7
Process 4530: value is 8
Process 4531: value is 8
Process 4530: value is 9
Process 4531: value is 9
```

*What determines the order in which
the two processes run???*

Why have fork() at all?

Why make a copy of the parent process?

Don't you usually want to start a new program instead?

Where might “cloning” the parent be useful?

-
-
-

Why have fork() at all?

Why make a copy of the parent process?

Don't you usually want to start a new program instead?

Where might “cloning” the parent be useful?

- Web server – make a copy for each incoming connection
- Parallel processing – set up initial state, fork off multiple copies to do work

UNIX philosophy: System calls should be minimal.

- Don't overload system calls with extra functionality if it is not always needed.
- Better to provide a flexible set of simple primitives and let programmers combine them in useful ways.

UNIX fork mechanism

So, how do you tell the parent from the child?

- Return value of `fork()` is zero in the child, or child's PID in the parent.

```
int main(int argc, char **argv) {
    int child_pid = fork();

    if (child_pid == 0) {
        /* Running in the child process */
        printf("I am the child process, parent process
                is %d\n", getppid());
        /* ... Do stuff for child ... */
    } else {
        /* Running in the parent process */
        printf("My child process is %d\n", child_pid);
        /* ... Do stuff for parent ... */
    }
}
```

Waiting for children

Child process runs “in parallel” with the parent

- Use the `wait()` system call to cause parent to wait for it.

```
int main(int argc, char **argv) {
    int i, rv;
    for (i = 0; i < 10; i++) {
        printf("Process %d: value is %d\n", getpid(), i);

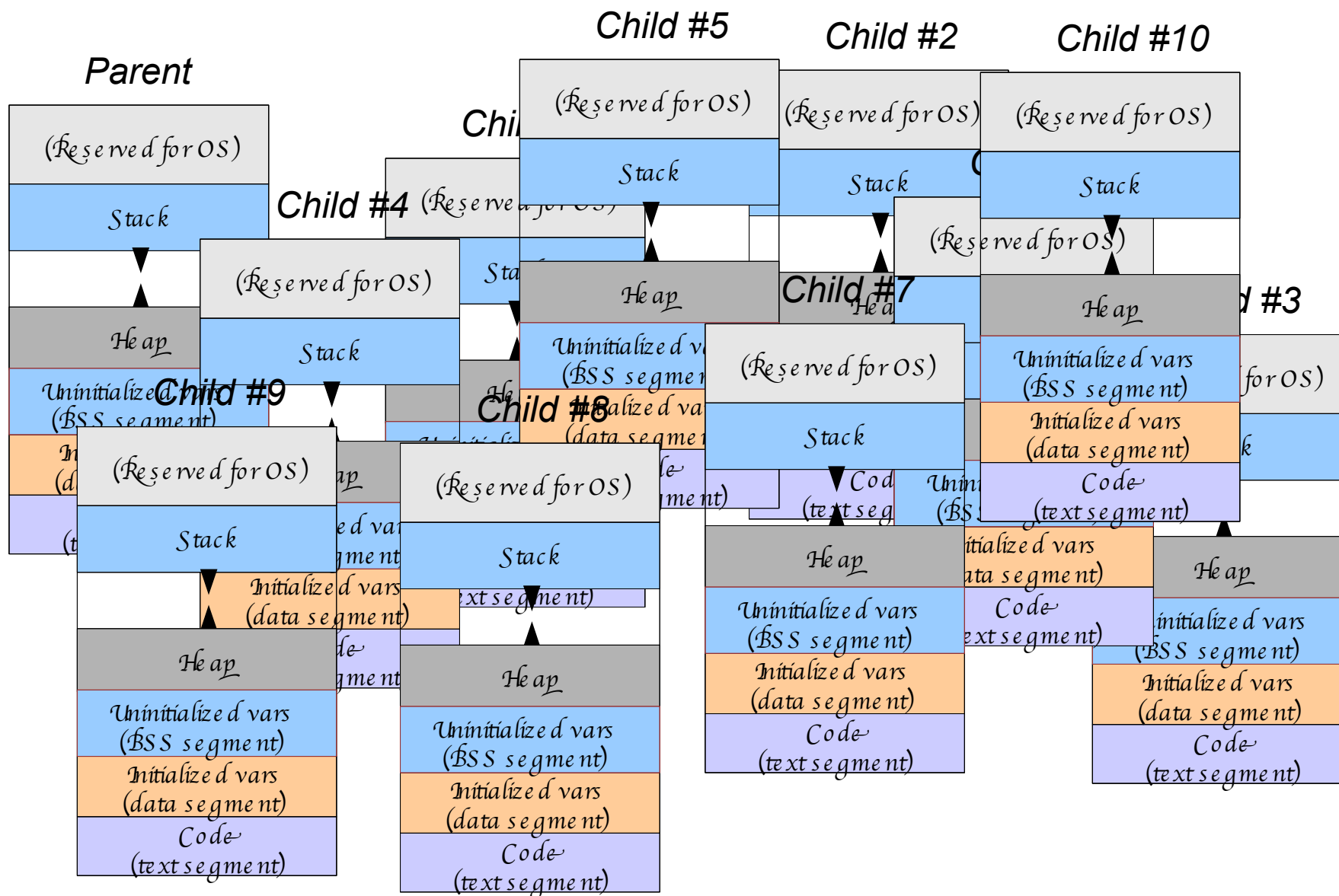
        if (i == 5) {
            printf("Process %d: About to do a fork...\n",
                getpid());
            int child_pid = fork();
            if (child_pid) wait(&rv);
            /* Only the parent waits here */
        }
    }
    exit(1);
    /* Return the exit value to the parent process */
}
```

Output of program with wait()

```
Process 11476: value is 0
Process 11476: value is 1
Process 11476: value is 2
Process 11476: value is 3
Process 11476: value is 4
Process 11476: value is 5
Process 11476: About to do a fork... ← Parent waits here...
Process 11477: value is 6
Process 11477: value is 7
Process 11477: value is 8
Process 11477: value is 9
Process 11476: value is 6
Process 11476: value is 7
Process 11476: value is 8
Process 11476: value is 9
```

Memory concerns

So fork makes a copy of a process. What about memory usage?





Memory concerns

OS aggressively tries to share memory between processes.

- Especially processes that are fork()'d copies of each other

Copies of a parent process do not actually get a private copy of the address space...

- ... Though that is the illusion that each process gets.
- Instead, they share the same physical memory, until one of them makes a change.

The virtual memory system is behind these shenanigans.

- We will discuss this in much detail later in the course

fork() and execve()

How do we start a new program, instead of just a copy of the old program?

- Use the UNIX `execve()` system call

```
int execve(const char *filename,  
           char *const argv [], char *const envp[]);
```

- `filename`: name of executable file to run
- `argv`: Command line arguments
- `envp`: environment variable settings (e.g., `$PATH`, `$HOME`, etc.)

fork() and execve()

execve() does not fork a new process!

- Rather, it **replaces the address space and CPU state of the current process**
- Loads the new address space from the executable file and starts it from main()
- So, to start a new program, use **fork()** followed by **execve()**



Using fork and exec

```
int main(int argc, char **argv) {
    int rv;
    if (fork()) {      /* Parent process */
        wait(&rv);
    } else {          /* Child process */
        char *newargs[3];
        printf("Hello, I am the child process.\n");
        newargs[0] = "/bin/echo"; /* Convention! Not required!! */
        newargs[1] = "some random string";
        newargs[2] = NULL;      /* Indicate end of args array */
        if (execv("/bin/echo", newargs)) {
            printf("warning: execve returned an error.\n");
            exit(-1);
        }
        printf("Child process should never get here\n");
        exit(42);
    }
}
```

Output of fork/exec example

Admin Stuff

Next Lecture: Threads

- Multiple CPU states in a single process

Read Tanenbaum 2.2

Don't forget about Assignment 0!!

- due **NEXT TUESDAY IN CLASS**