Scheduling

Daniel Hagimont (INPT)

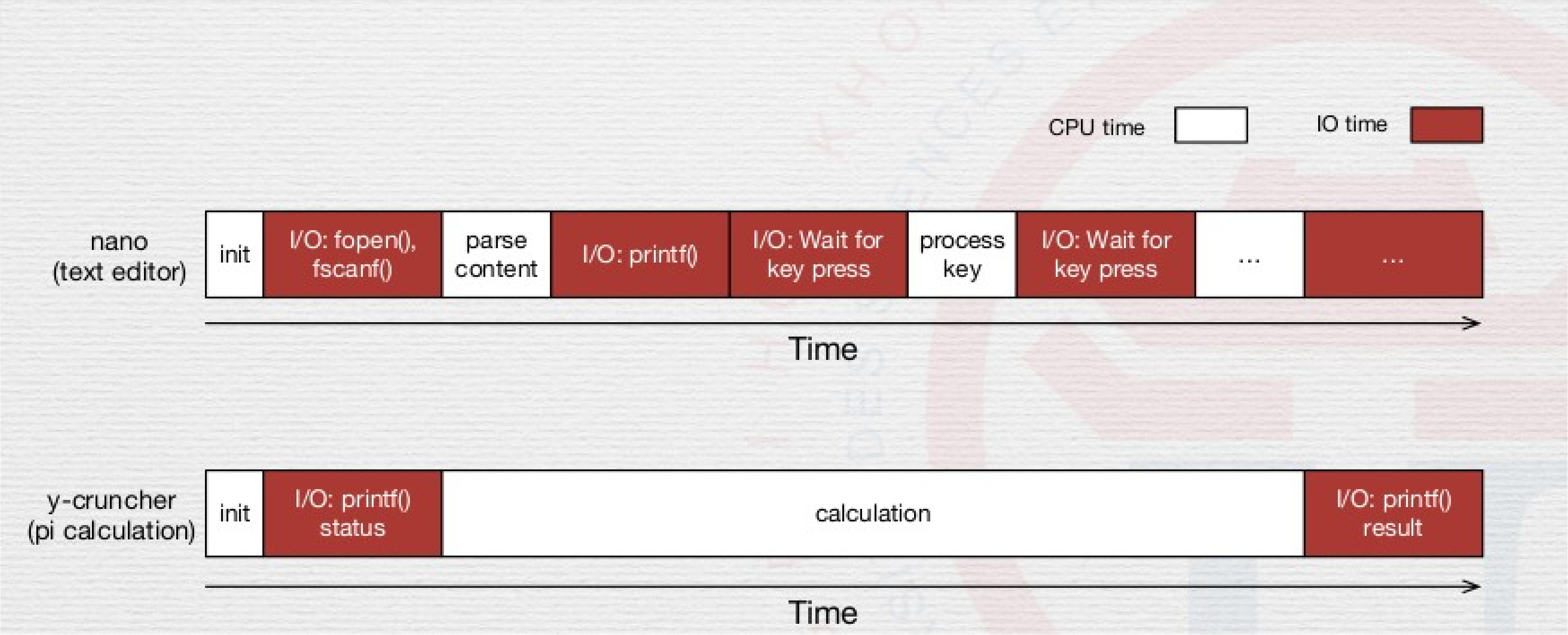
hagimont@enseeiht.fr

http://hagimont.perso.enseeiht.fr

Scheduler

- The kernel component that
  - Select a process to be executed on a CPU
- Maximize CPU usage
  - For a set of process
  - With one or more CPU
- Different characteristics of processes
  - CPU bound: spend more time on computation
  - I/O bound: spend more time on I/O devices (reading/writing on disk …)
- Process execution consists of
  - CPU execution
  - I/O wait

Task scheduling

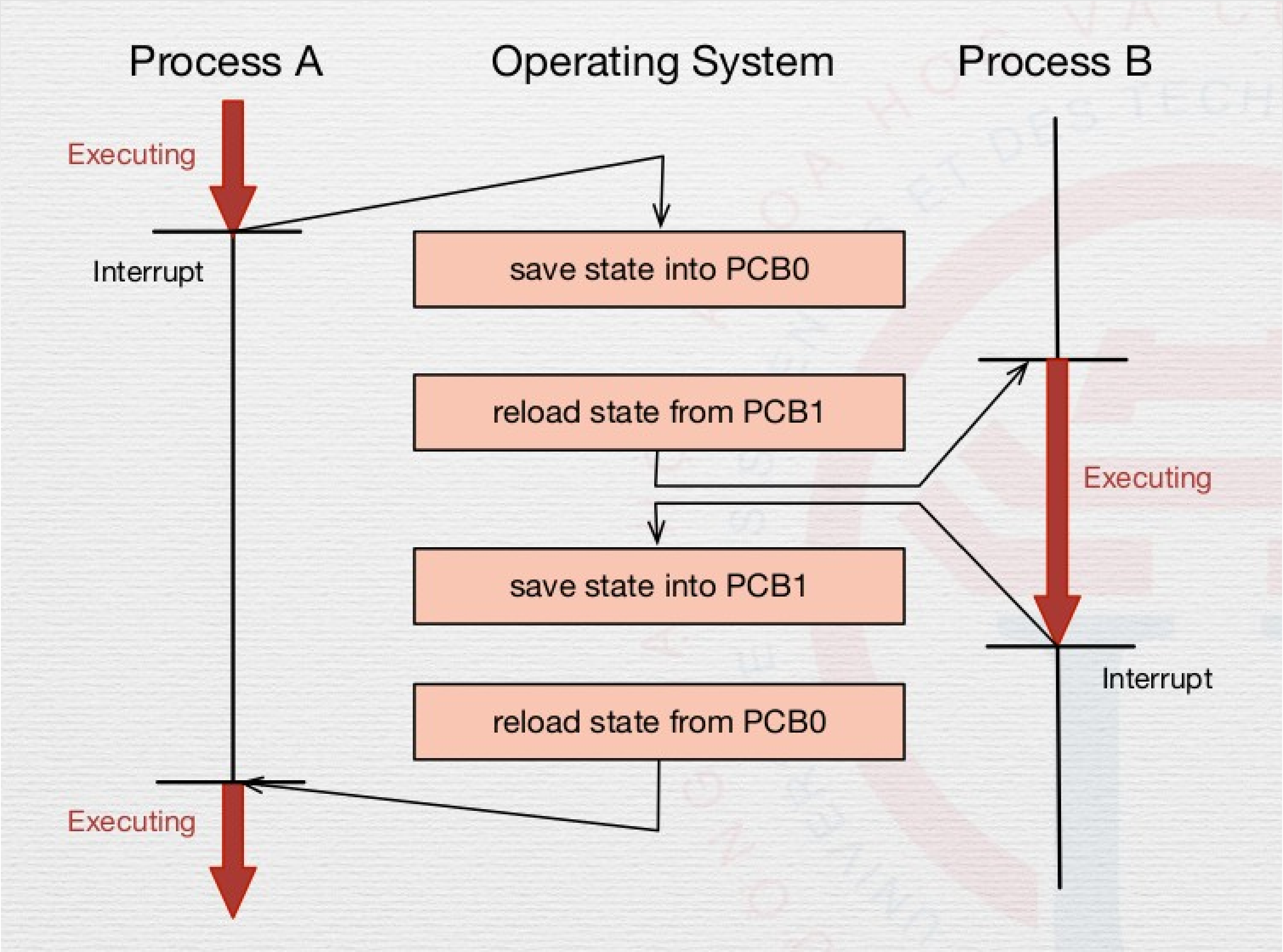


Different tasks with different priorities

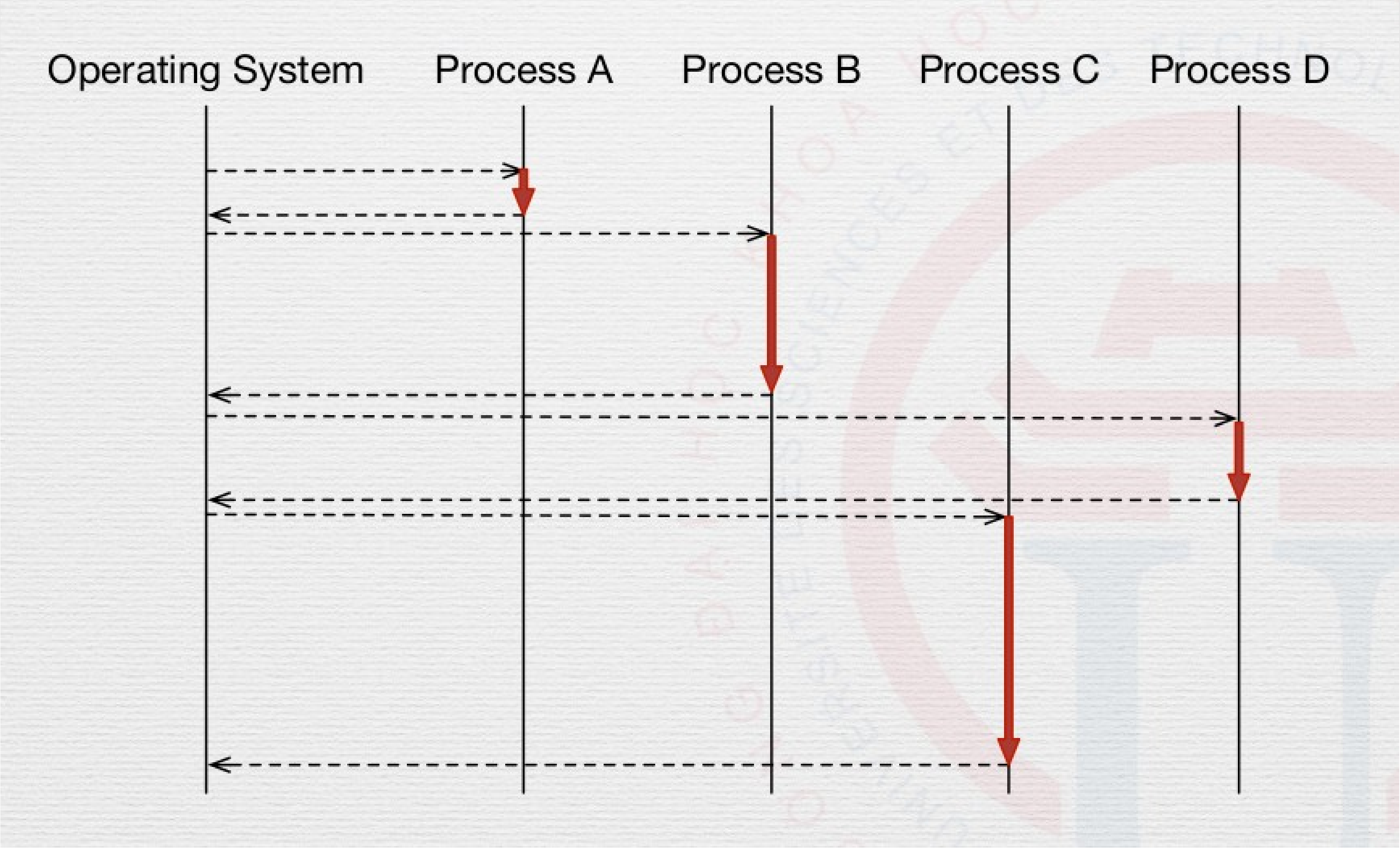
Characteristics of schedulers

- Ability to pause running processes
  - Preemption: OS forcely pauses running processes
  - Non-preemption: only at the end of tasks or process willing to pause itself
- Duration between each switch
  - Short term scheduler: milliseconds (fast, responsive)
  - Long term scheduler: seconds/minutes (batch jobs)
- Switch between processes
  - Save data of old process
  - Load previously saved data of new process
  - Context switch is overhead

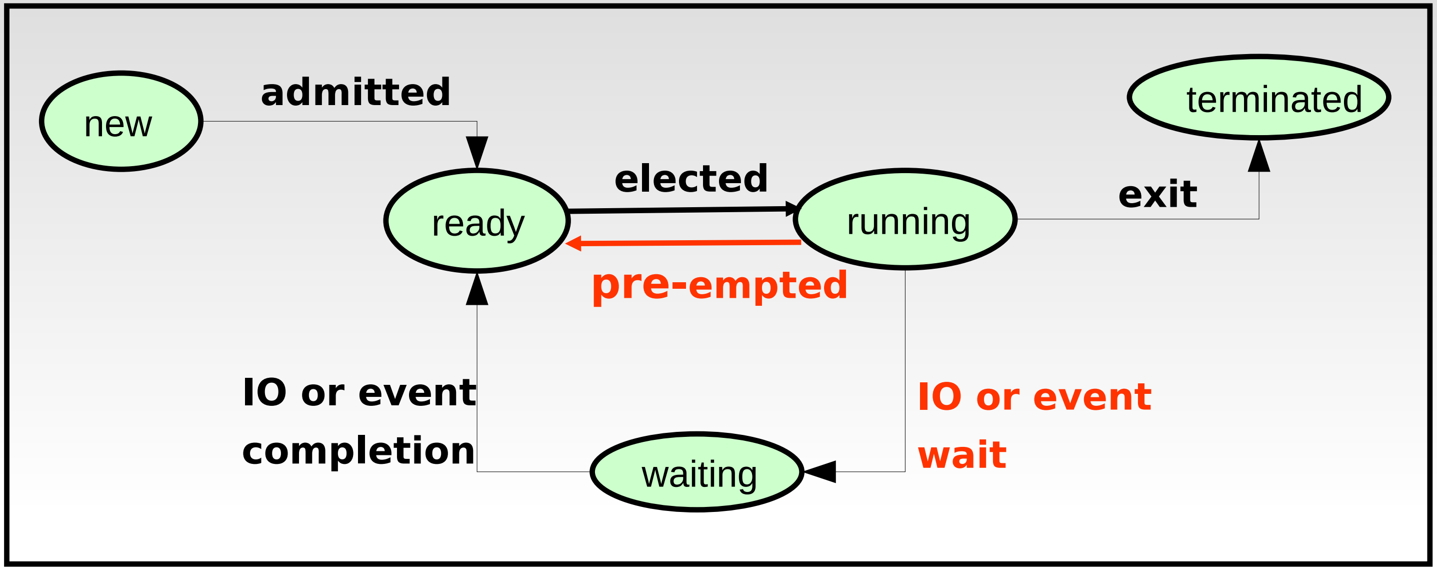
Context switches



Context switches



Context switches



- When to perform a context switch
  - Process switches from running to waiting (IO) – non preemptive
  - Process terminates (exit) – non preemptive
  - Process switches from running to ready – preemptive
  - Process switches from witing to ready - preemptive

Process management by the OS

- Process queues
  - Ready queue (ready processes)
  - Device queue (Process waiting for IO)
  - Blocked Queue (Process waiting for an event)
  - ...
- OS migrates processes across queues

CPU Allocation to processes

- The scheduler is the OS's part that manages CPU allocation
- Criteria / Scheduling Algorithm
  - Fair (no starvation)
  - Minimize the waiting time for processes
  - Maximize the efficiency (number of jobs per unit of time)

Simple scheduling algorithms (1/2)

- Non-pre-emptive scheduler
  - FCFS (First Come First Served)
    - Fair, maximize efficiency
- Pre-emptive scheduler
  - SJF (Shortest Job First)
    - Priority to shortest task
    - Require to know the execution time (model estimated from previous execution)
    - Unfair but optimal in term of response time
  - Round Robin (fixed quantum)
    - Each processus is affected a CPU quantum (10-100 ms) before pre-emption
    - Efficient (unless the quantum is too small), fair / response time (unless the quantum too long)

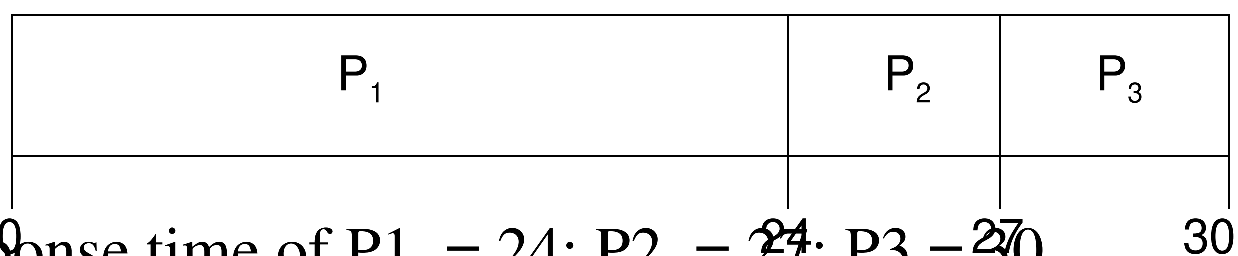
Simple scheduling algorithms (2/2)

- Round robin with dynamic priority
  - A priority associated with each process
  - One ready queue per priority level
  - Decrease priority for long tasks (prevent starvation)
- Round robin with variable quantum
  - N^{th} scheduling receives 2^{N-1} quantum (reduce context-switches)

First-Come, First-Served (FCFS) non pre-emptive (1/2)

Process's	execution time
P1	24
P2	3
P3	3

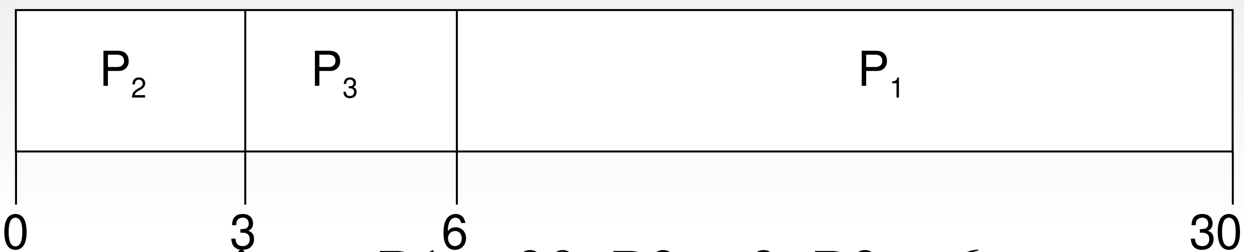
- Let's these processes come in this order : P1,P2,P3



- Response time of P1 = 24; P2 = 27; P3 = 30
- Mean time : (24 + 27 + 30)/3 = 27

First-Come, First-Served (FCFS) (2/2)

- Let's these processes come in this order : P2 , P3 , P1 .



- Response time : P1 = 30; P2 = 3; P3 = 6
- Mean time : (30 + 3 + 6)/3 = 13
- Better than the previous case
- Schedule short processes before

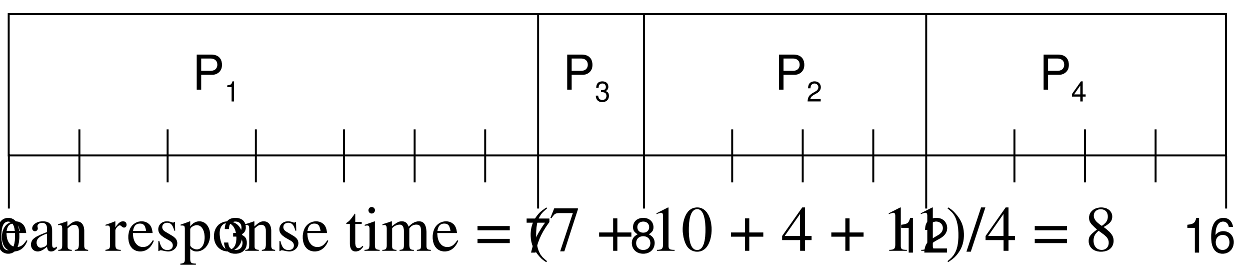
Shortest-Job-First (SJF)

- Associate with each process its execution time
- Two possibilities :
  - Non pre-emptive – When a CPU is allocated to a process, it cannot be pre-empted
  - Pre-emptive – if a new process comes with a shorter execution time than the running one, this last process is pre-empted (Shortest-Remaining-Time-First - SRTF)
- SJF is optimal / mean response time

Non Pre-emptive SJF

Process	Come in	Exec. Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (non pre-emptive)

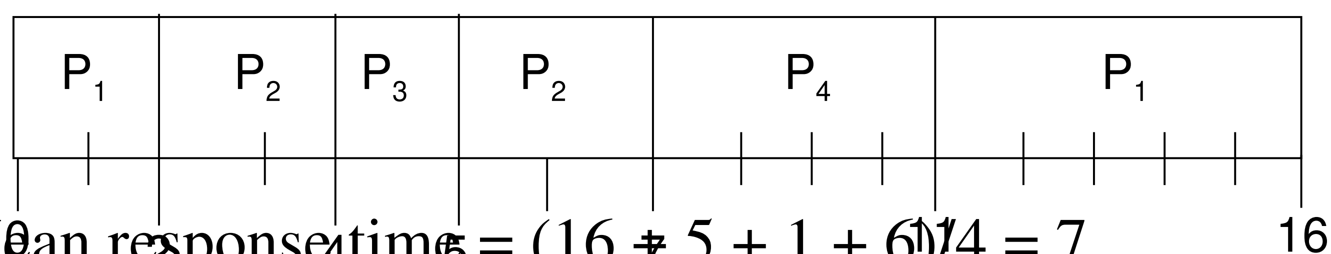


- Mean response time = (7 + 10 + 4 + 11)/4 = 8

Pre-emptive SJF (SRTF)

Process	Come in	Exec. Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (pre-emptive)

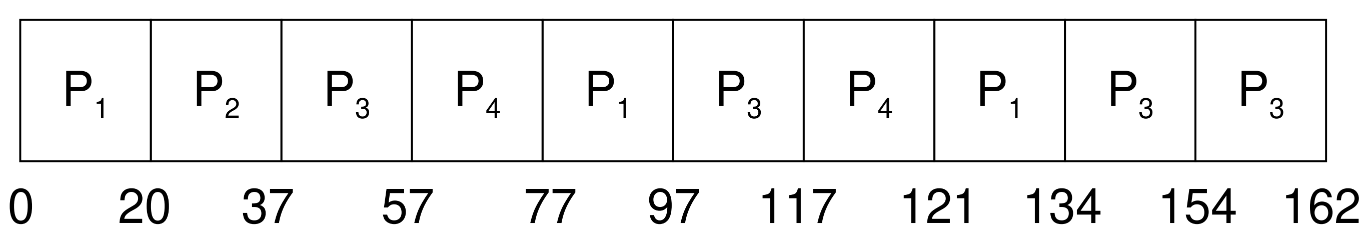


- Mean response time = (16 + 5 + 1 + 6)/4 = 7

Round Robin (Quantum = 20ms)

Process	Exec Time
P1	53
P2	17
P3	68
P4	24

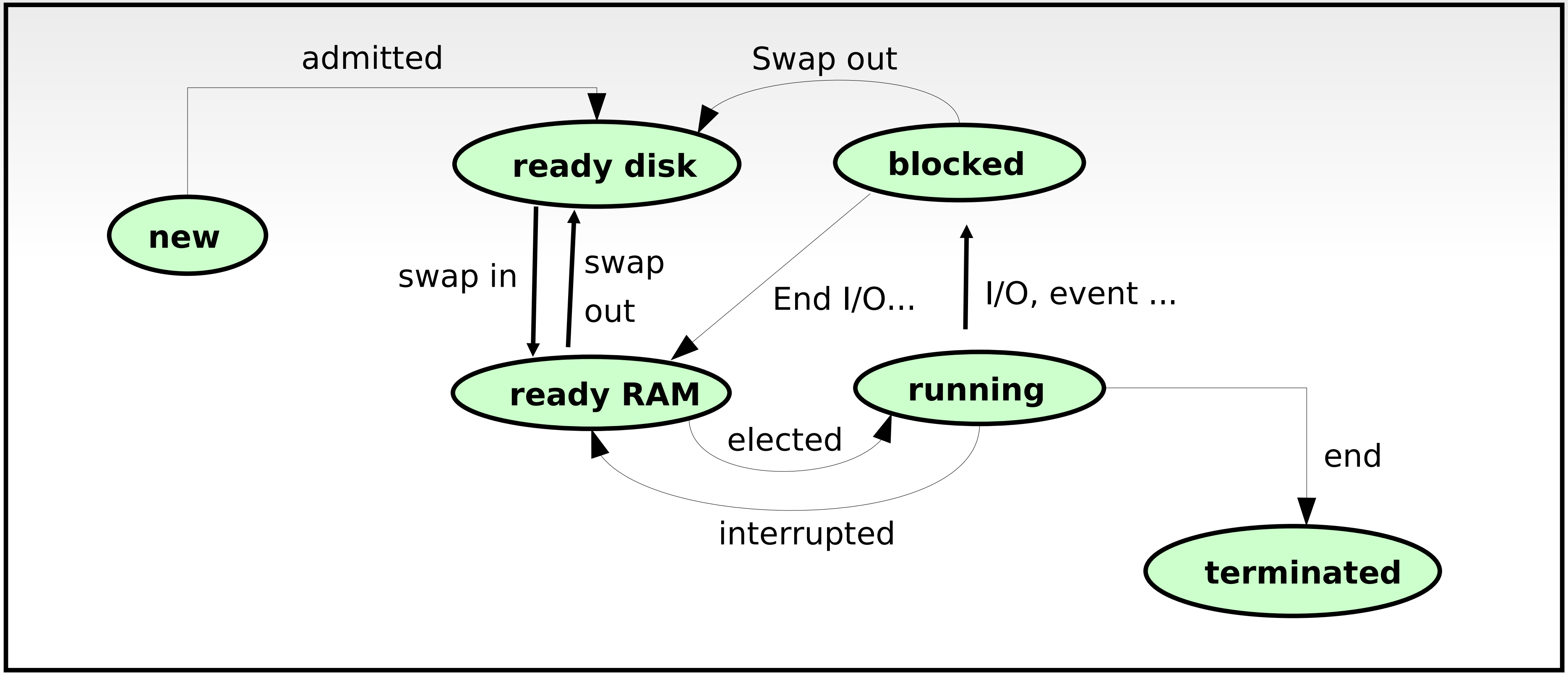
- Efficiency and mean response worse than SJF
- But don't need to estimate execution time



Multiple level scheduling algorithm

- The set of ready processes too big to fit in memory
- Part of these processes are swapped out to disk. This increases their activation time
- The elected process is always choosen from those that are in memory
- In parallel, another scheduling algorithm is used to manage the migration of ready process between disk and memory

Two level scheduling



A scheduler implementation (sched)

```
void thread0() {
	int i,k;
	for (i=0;i<10;i++) {
		printf("thread 0\n");
		sleep(1);
	}
}
void thread1() {
	int i,k;
	for (i=0;i<10;i++) {
		printf("thread 1\n");
		sleep(1);
	}
}
void thread2() {
	int i,k;
	for (i=0;i<10;i++) {
		printf("thread 2\n");
		sleep(1);
	}
}
```

A scheduler implementation (sched)

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <stdlib.h>
#include <ucontext.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define MAX_THREAD 3
#define STACK_SIZE 16000
#define TIME_SLICE 4

void thread0();
void thread1();
void thread2();
void schedule(int sig);

ucontext_t uctx_main;

void (*thread_routine[MAX_THREAD])() = {thread0, thread1, thread2};
ucontext_t thread_save[MAX_THREAD];
char thread_stack[MAX_THREAD][STACK_SIZE];
int thread_state[MAX_THREAD];
int current;
```

A scheduler implementation (sched)

```
int main() {
	int i;
	for (i=0;i<MAX_THREAD;i++) {
		if (getcontext(&thread_save[i]) == -1)
			{ perror("getcontext"); exit(0); }
		thread_save[i].uc_stack.ss_sp = thread_stack[i];
		thread_save[i].uc_stack.ss_size = sizeof(thread_stack[i]);
		thread_save[i].uc_link = &uctx_main;
		makecontext(&thread_save[i], thread_routine[i], 0);
		thread_state[i] = 1;
		printf("main: thread %d created\n",i);
	}

	signal(SIGALRM, schedule);
	alarm(TIME_SLICE);

	printf("main: swapcontext thread 0\n");
	current = 0;
	if (swapcontext(&uctx_main, &thread_save[0]) == -1)
		{ perror("swapcontext"); exit(0); }

	while (1) {
		printf("thread %d completed\n", current);
		thread_state[current] = 0;
		schedule(0);
	}
}
```

A scheduler implementation (sched)

```
void schedule(int sig) {
        int k, old;
        alarm(TIME_SLICE);
        old = current;
        for (k=0;k<MAX_THREAD;k++) {
                current = (current + 1) % MAX_THREAD;
                if (thread_state[current] == 1) break;
        }
        if (k==MAX_THREAD) {
                printf("last thread completed: exiting\n");
                exit(0);
        }
        printf("schedule: save(%d) restore (%d)\n",old, current);
        if (swapcontext(&thread_save[old], &thread_save[current]) == -1)
                { perror("swapcontext"); exit(0); }
}
```

Exercise (sched)

```
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Systemes/scheduler$ gcc sched
-ctx.c -o sched
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Systemes/scheduler$ ./sched
main: thread 0 created
main: thread 1 created
main: thread 2 created
main: swapcontext thread 0
thread 0
thread 0
thread 0
thread 0
schedule: save(0) restore (1)
thread 1
thread 1
thread 1
thread 1
schedule: save(1) restore (2)
thread 2
thread 2
thread 2
thread 2
schedule: save(2) restore (0)
thread 0
thread 0
thread 0
thread 0
schedule: save(0) restore (1)
thread 1
thread 1
thread 1
^C
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Systemes/scheduler$
```

Resources you can read

- Operating System Concepts, 10th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne
  - http://os-book.com/
  - Chapters 5
- Modern Operating Systems, Andrew Tanenbaum
  - http://www.cs.vu.nl/~ast/books/mos2/
  - Chapter 2 (2.5)