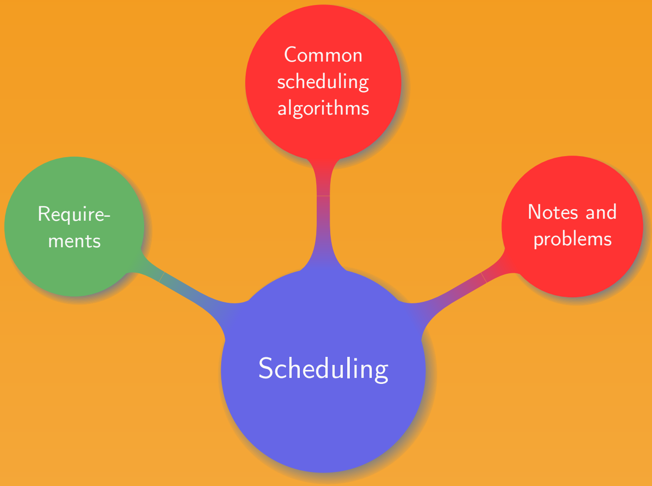
Introduction to Operating Systems

4. Scheduling

Manuel – Fall 2019

Chapter organisation



Initial problem

Scheduler's job:

- Multiple processes competing for using the CPU
- More than one process in ready state
- Which one to select next?
- Key issue in terms of "perceived performance"
- Need "clever" and efficient scheduling algorithms

Scheduler's job

Switching process is expensive:

- Switch from user mode to kernel mode
- Save state of current process (save register, memory map...)
- Run scheduling algorithm to select a new process
- Remap the memory address for the new process (convert address generated by the program into physical memory address in RAM); done by the Memory Management Unit (MMU)
- Start new process

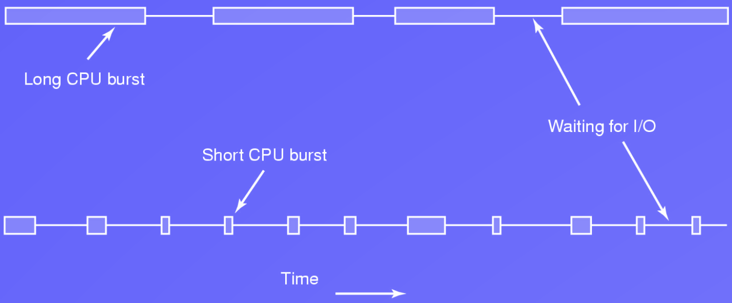
Scheduler's job

Switching process is expensive:

- Switch from user mode to kernel mode
- Save state of current process (save register, memory map...)
- Run scheduling algorithm to select a new process
- Remap the memory address for the new process (convert address generated by the program into physical memory address in RAM); done by the Memory Management Unit (MMU)
- Start new process

Remark.
Too many switches per second wastes much CPU

Process behavior



Typical behavior:

- Process runs for a while
- System call emitted to read/write from/in a file
- More general: process in blocked state until external device has completed its work

Process behavior

Compute bound vs. I/O bound:

- Most time spent computing vs. waiting for I/O
- Length of the CPU burst:
  - I/O time is constant
  - Processing data is not constant
- As CPUs get faster processes are more and more I/O bound

Process behavior

Compute bound vs. I/O bound:

- Most time spent computing vs. waiting for I/O
- Length of the CPU burst:
  - I/O time is constant
  - Processing data is not constant
- As CPUs get faster processes are more and more I/O bound

Conclusion: I/O bound processes should be run quickly, such as to issue their I/O request and keep the disk busy

When to schedule?

Basic scheduling issues:

- Creation of a new process: both the parent and the child are in ready state
- A process exits: which process to run next?
  An idle process is run if no process is ready
- A process blocks (semaphore, I/O...), what process to run?
  Note: a blocked process could be waiting for another process, how to run this specific process?
- I/O interrupt from a device that has completed its task: run the newly ready process, or another one?

Preemptive vs. non-preemptive

Two main strategies for scheduling algorithms:

1. Preemptive: a process is run for at most *n* ms; if it is not done at the end of the period then it is suspended and another process is selected and run: require a clock interrupt

2. Non-preemptive: a process runs until it blocks or voluntarily releases the CPU; it is resumed after an interrupt unless another process with higher priority is in the queue

What is the goal?

All systems:

- Fairness: fair share of the CPU for each process
- Policy enforcement: following the defined policy
- Balance: all parts of the system are busy

What is the goal?

All systems:

- Fairness: fair share of the CPU for each process
- Policy enforcement: following the defined policy
- Balance: all parts of the system are busy

Batch systems:

- Throughput: maximise the number of jobs per hour
- Turnaround time: minimise the time between submission and termination of a job
- CPU utilisation: keep the CPU as busy as possible

What is the goal?

Interactive systems:

- Response time: quickly respond to requests
- Proportionality: meet user's expectations

What is the goal?

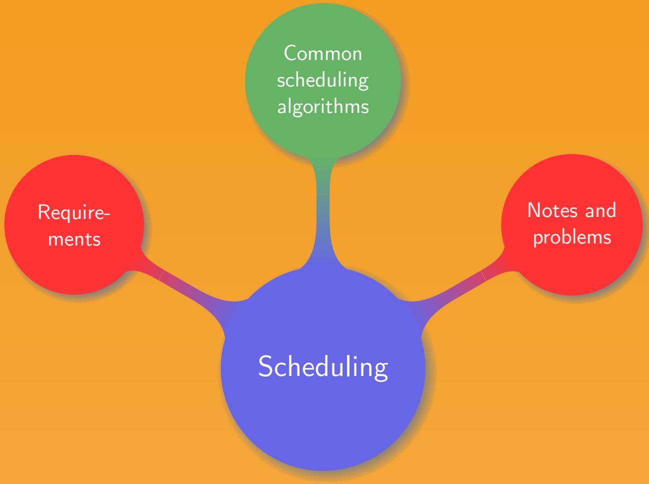
Interactive systems:

- Response time: quickly respond to requests
- Proportionality: meet user's expectations

Real-time systems:

- Meeting deadlines: avoid losing data
- Predictability: avoid quality degradation for multimedia

Chapter organisation



First-come, first-served

Basics: simplest algorithm, non-preemptive

- CPU is assigned in the order it is requested
- Processes are not interrupted, they can run a long as they want
- New jobs are put at the end of the queue
- When a process blocks the next in line is run
- When a blocked process becomes ready it is put at the end of the queue

First-come, first-served

Basics: simplest algorithm, non-preemptive

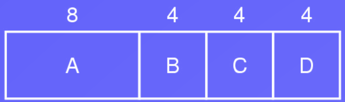
- CPU is assigned in the order it is requested
- Processes are not interrupted, they can run a long as they want
- New jobs are put at the end of the queue
- When a process blocks the next in line is run
- When a blocked process becomes ready it is put at the end of the queue

Characteristics: good for batch systems, implemented using a single linked list

Drawback: what happens when there is one compute-bound process and many I/O-bound ones?

Shortest job first

Basics: Non-preemptive, run times are known in advance

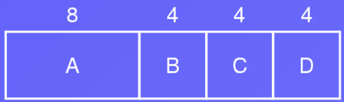


- Run time: A: 8 min, B: 4 min, C: 4 min, D: 4 min
- Turnaround time: $\frac{8+12+16+20}{4} = 14$ min

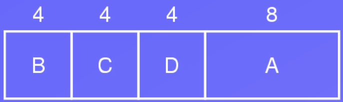
- Run time: B: 4 min, C: 4 min, D: 4 min, A: 8 min
- Turnaround time: $\frac{4+8+12+20}{4} = 11$ min

Shortest job first

Basics: Non-preemptive, run times are known in advance



- Run time: A: 8 min, B: 4 min, C: 4 min, D: 4 min
- Turnaround time: $\frac{8+12+16+20}{4} = 14$ min



- Run time: B: 4 min, C: 4 min, D: 4 min, A: 8 min
- Turnaround time: $\frac{4+8+12+20}{4} = 11$ min

Characteristics: good for batch systems, very specific to a system

Drawback: what happens when the jobs are not all in a queue at the beginning?

Round-Robin scheduling

Basics: preemptive, old, simple, fair, most widely used

- Each process is assigned a time interval called *quantum*
- A process runs until (i) it blocks, (ii) it is finished or (iii) its quantum is over
- Switching process is then done

Round-Robin scheduling

Basics: preemptive, old, simple, fair, most widely used

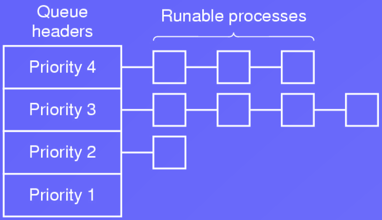
- Each process is assigned a time interval called *quantum*
- A process runs until (i) it blocks, (ii) it is finished or (iii) its quantum is over
- Switching process is then done

Characteristics: good for interactive systems, easy to implement (only needs to maintain a list of runnable processes)

Drawback: how long should a quantum be?

Priority scheduling

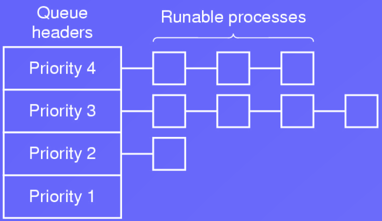
Basics: preemptive, priority depending on who or what runs



- Processes are more or less important (e.g. printing...)
- Creates priority classes
- Use Round-Robin within a class
- Run higher priority processes first

Priority scheduling

Basics: preemptive, priority depending on who or what runs



- Processes are more or less important (e.g. printing...)
- Creates priority classes
- Use Round-Robin within a class
- Run higher priority processes first

Characteristics: good for interactive systems, flexible, adjustable

Drawback: what happens if many high-priority processes run for a long time?

Lottery scheduling

Basics: preemptive, extends priority scheduling

- Processes get “lottery tickets”
- When a scheduling decision is made a random ticket is chosen
- Price for the winner is to access resources
- High priority processes get more tickets (higher probability of winning)

Lottery scheduling

Basics: preemptive, extends priority scheduling

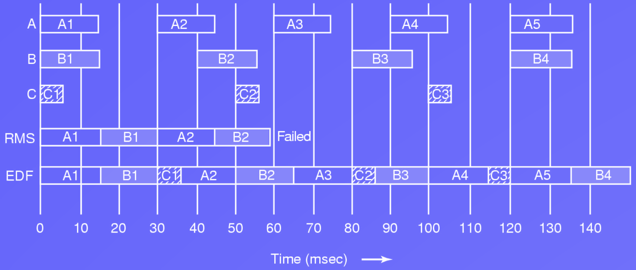
- Processes get "lottery tickets"
- When a scheduling decision is made a random ticket is chosen
- Price for the winner is to access resources
- High priority processes get more tickets (higher probability of winning)

Characteristics: good for interactive systems, highly responsive, possibility of cooperation between processes

Drawback: what if a "low level user" runs many small processes instead of a big one?

Earliest deadline first

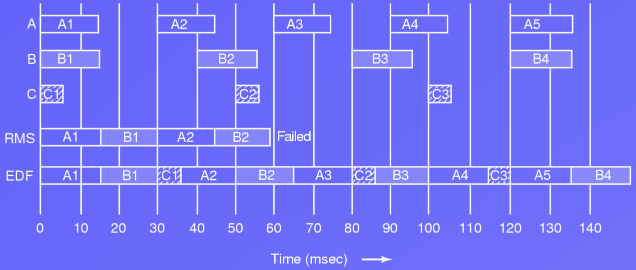
Basics: preemptive, priority based



- Process needs to announce (i) its presence and (ii) its deadline
- Scheduler orders processes with respect to their deadline
- First process in the list (earliest deadline) is run

Earliest deadline first

Basics: preemptive, priority based

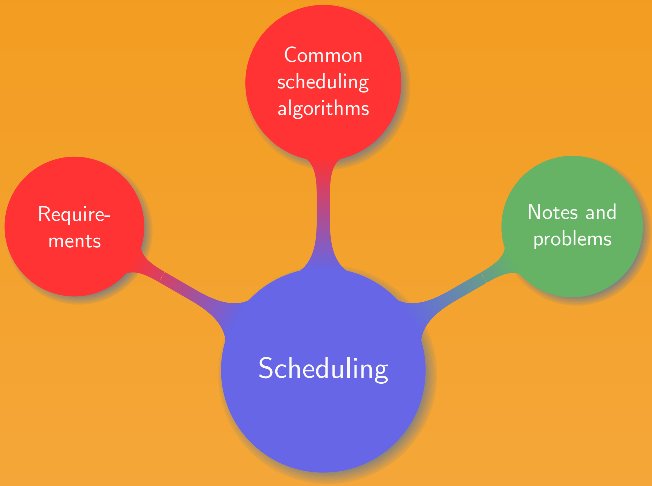


- Process needs to announce (i) its presence and (ii) its deadline
- Scheduler orders processes with respect to their deadline
- First process in the list (earliest deadline) is run

Characteristics: good for realtime systems, can fully use the CPU

Drawback: complex to implement

Chapter organisation



Policy vs. mechanism

Limitations of the previous algorithms:

- They all assume that processes are competing
- Parent could know which of its children is most important

Policy vs. mechanism

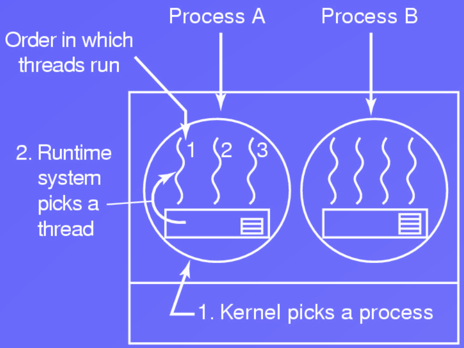
Limitations of the previous algorithms:

- They all assume that processes are competing
- Parent could know which of its children is most important

The solution consists in separating the scheduling mechanism from the scheduling policy:

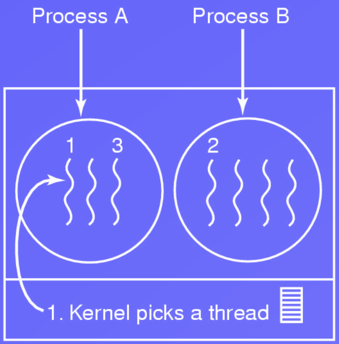
- Scheduling algorithm has parameters
- Parameters can be set by processes
- A parent can decide which of its children should have higher priority

Threads scheduling – User-level



Possible: A1, A2, A3, A1, A2, A3
Impossible: A1, B1, A2, B2, A3, B3

Threads scheduling – Kernel-level



Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

The dining philosophers problem

Synchronisation problem:

- A philosopher is either thinking or eating
- When he is hungry he takes:
  1. His left chop-stick
  2. His right chop-stick
- Eats
- Puts down his chop-sticks
- Thinks

The dining philosophers problem

First obvious solution:

- Wait for a chop-stick to be available
- Seize it as soon as it becomes available

The dining philosophers problem

First obvious solution:

- Wait for a chop-stick to be available
- Seize it as soon as it becomes available

Problem: they all take the left chop-stick at the same time and wait forever for the right one

The dining philosophers problem

First obvious solution:

- Wait for a chop-stick to be available
- Seize it as soon as it becomes available

Problem: they all take the left chop-stick at the same time and wait forever for the right one

Second solution:

- Take left chop-stick
- If right chopstick not available put down the left one
- Wait for some time and repeat the process

The dining philosophers problem

First obvious solution:

- Wait for a chop-stick to be available
- Seize it as soon as it becomes available

Problem: they all take the left chop-stick at the same time and wait forever for the right one

Second solution:

- Take left chop-stick
- If right chopstick not available put down the left one
- Wait for some time and repeat the process

Problem: all process start at the same time, then nobody ever eats

The dining philosophers problem

What about using mutex?

The dining philosophers problem

What about using mutex?

- Philosopher thinks
- Lock mutex
- Acquire chop-sticks, eat, put down chop-sticks
- Unlock mutex

The dining philosophers problem

What about using mutex?

- Philosopher thinks
- Lock mutex
- Acquire chop-sticks, eat, put down chop-sticks
- Unlock mutex

Problem: how many philosophers can eat at the same time?

The dining philosophers problem

```
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
enum { THINKING, HUNGRY, EATING };
int state[N]; mutex mut = 0 ; semaphore s[N];
void philosopher(int i) {while(TRUE) {think();take_cs(i);eat();put_cs(i);}}
void take_cs(int i) {
  mutex-lock(&mut);
  state[i] = HUNGRY; test(i);
  mutex-unlock(&mut); down(&s[i]);
}
void put_cs(int i) {
  mutex-lock(&mut);
  state[i] = THINKING; test(LEFT); test(RIGHT);
  mutex-unlock(&mut);
}
void test(int i) {
  if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING;) {
    state[i]=EATING; up(&s[i]); }
}
```

Key points

- Why is scheduling the lowest part of the OS?
- What are the two main types of algorithm?
- What are the two most common scheduling algorithms?
- Give an example of theoretical problem related to scheduling

Thank you!