Process Scheduling

CS 241

March 5, 2014

Copyright © University of Illinois CS 241 Staff

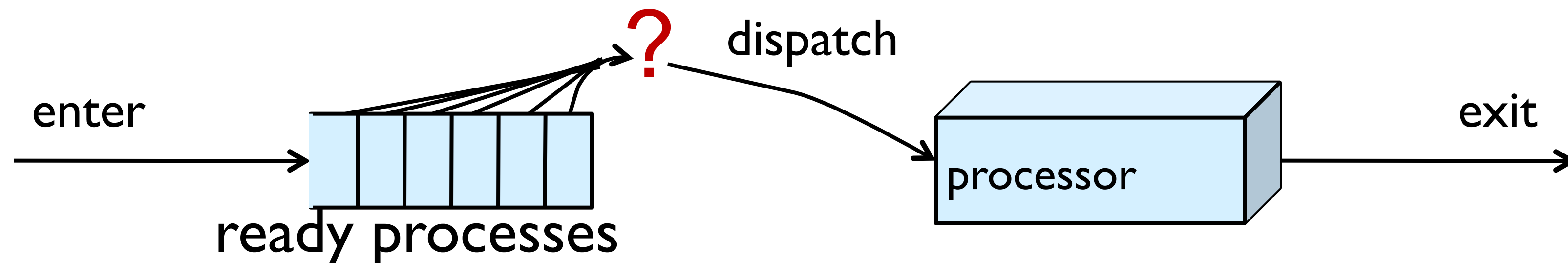
Process scheduling

Deciding which process/thread should occupy each resource (CPU, disk, etc.) at each moment

Scheduling is everywhere...

- disk reads
- process/thread resource allocation
- servicing clients in a web server
- compute jobs in clusters / data centers
- jobs using physical machines in factories

The basic scheduling decision



Given a set of ready processes

- Which one should I run next?
- How long should it run?
- ...for each resource (CPU, disk, ...)

Same underlying concepts apply to scheduling processes or threads

- or picking packets to send in routers
- or scheduling jobs in physical factories

Simplest scheduling algorithm: First Come First Serve (FCFS)

Process that requests the CPU first is allocated the CPU first

- Also called FIFO

Non-preemptive

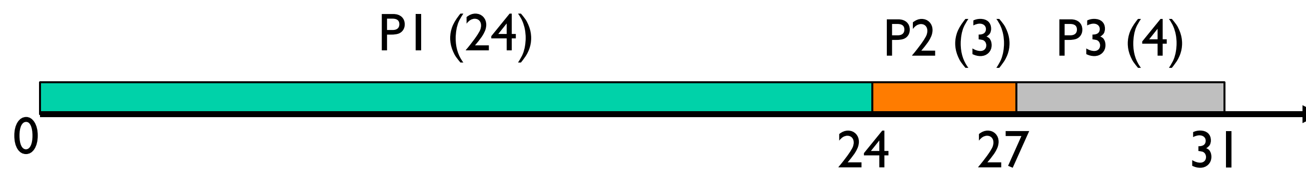
- Used in batch systems

Implementation

- FIFO queues
- A new process enters the tail of the queue
- The scheduler selects next process to run from the head of the queue

FCFS Example

Process	Duration	Order	Arrival Time
P1	24	1	0
P2	3	2	3
P3	4	3	7

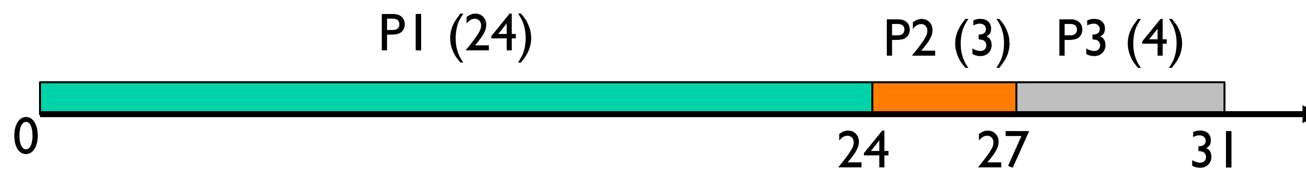


P1 waiting time:
P2 waiting time:
P3 waiting time:

The average waiting time:

FCFS Example

Process	Duration	Order	Arrival Time
P1	24	1	0
P2	3	2	3
P3	4	3	7



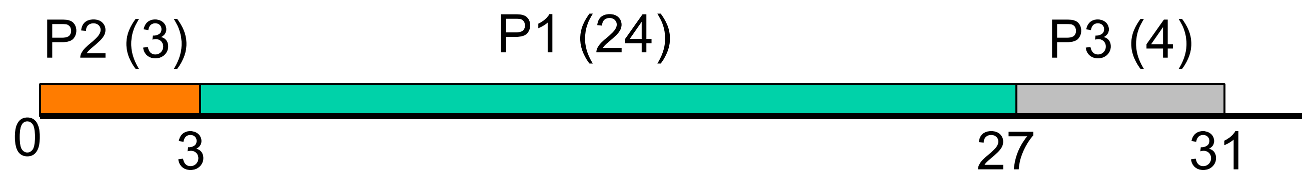
P1 waiting time: 0
P2 waiting time: 24-3=21
P3 waiting time: 27-7=20

The average waiting time:
(0+21+20)/3 = 13.67

FCFS Example

Process	Duration	Order	Arrival Time
P2	24	2	3
P1	3	1	0
P3	4	3	7

What if the arrival times of P1 and P2 are swapped?



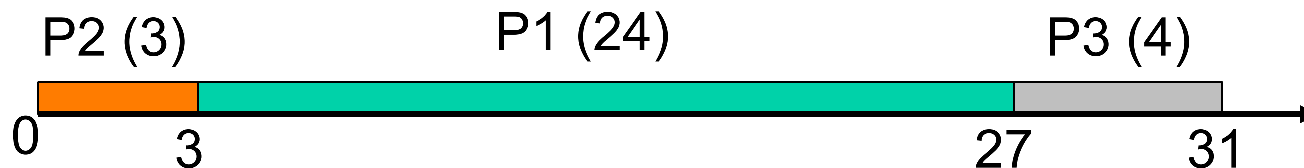
P1 waiting time:
P2 waiting time:
P3 waiting time:

The average waiting time:

FCFS Example

Process	Duration	Order	Arrival Time
P2	24	2	3
P1	3	1	0
P3	4	3	7

What if the arrival times of P1 and P2 are swapped?



P1 waiting time: 3-3=0
P2 waiting time: 0
P3 waiting time: 27-7=20

The average waiting time:
(0+0+20)/3=6.67

Problems with FCFS

Not optimal mean response time

- Schedule depends on order jobs happen to arrive
- How would you fix that?
  - Shortest Job First (best you can do without preemption)

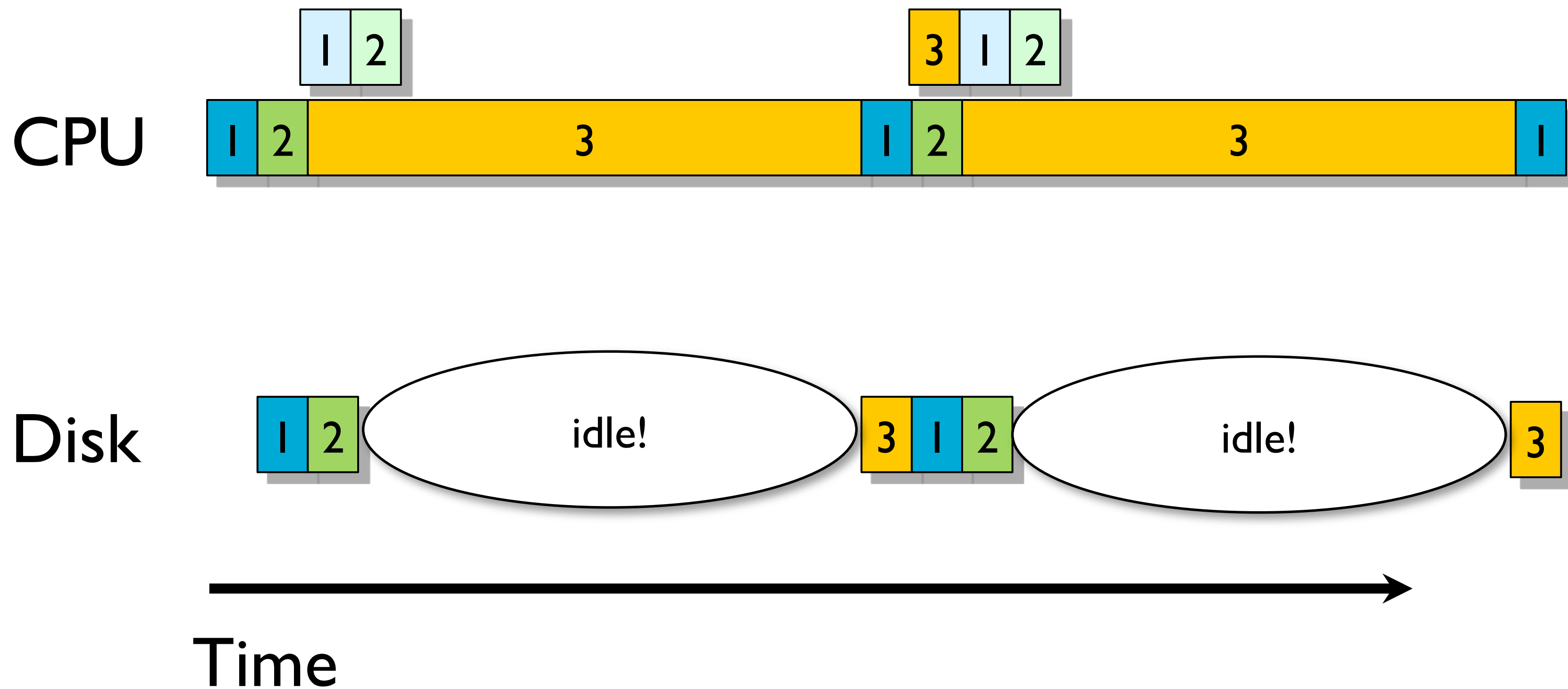
Long job may cause long wait for others

Poor parallelism

- May have low CPU and I/O device utilization

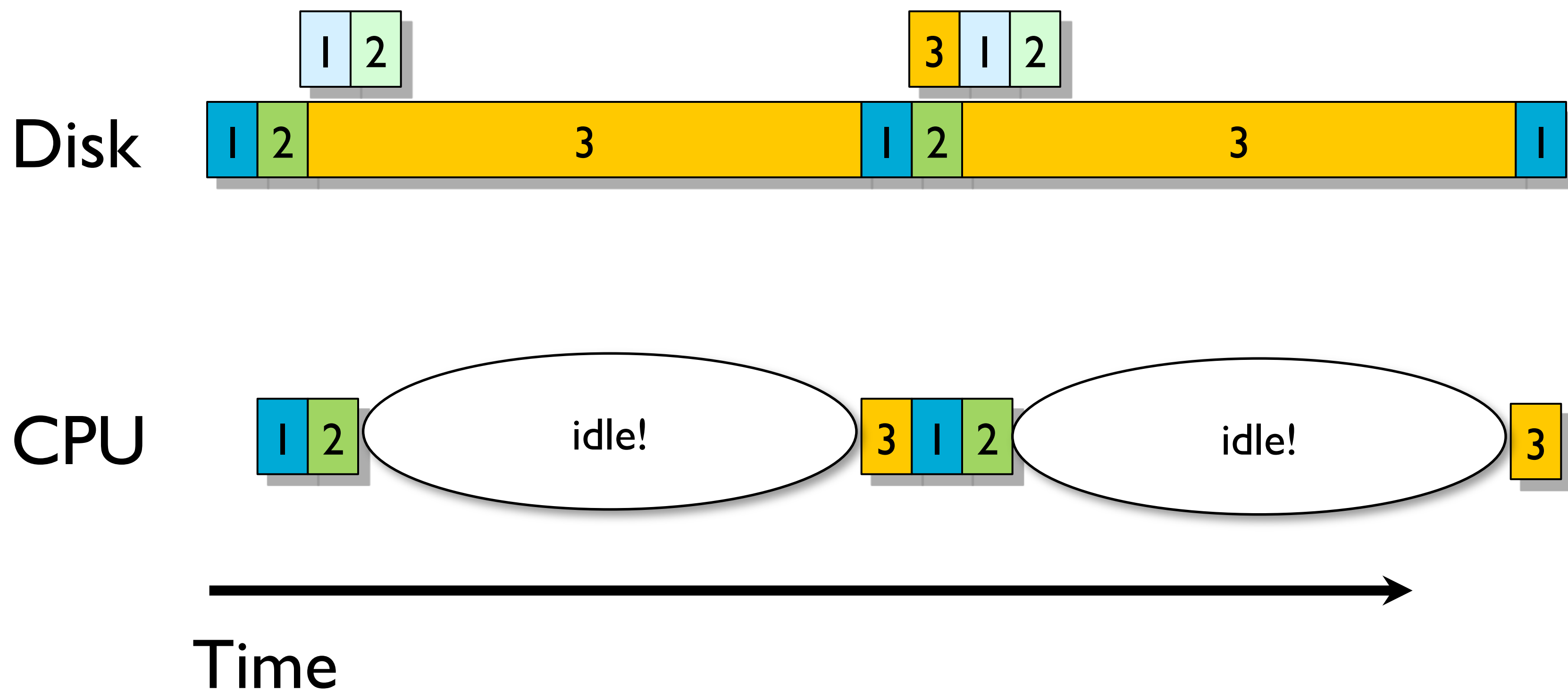
FCFS: poor parallelism

Jobs 1,2: a **msec** of CPU, a disk read, repeat
Job 3: a **sec** of CPU, a disk read, repeat



FCFS: poor parallelism

Jobs 1,2: a **msec** of disk, a little CPU, repeat
Job 3: a **sec** of disk, a little CPU, repeat



Thus far: Batch scheduling

FCFS, SJF useful when fast response not necessary

- weather simulation
- rendering an animated movie
- processing click logs to match advertisements with users
- ...

What if we need to respond to events quickly?

- playing frames of a movie
- human interacting with computer
- packets arriving every few milliseconds
- ...

Interactive Scheduling

Usually preemptive

- Time is sliced into quanta, i.e., time intervals
- Scheduling decisions are made at the beginning of each quantum

Performance metrics

- Average response time
- Fairness (or proportional resource allocation)

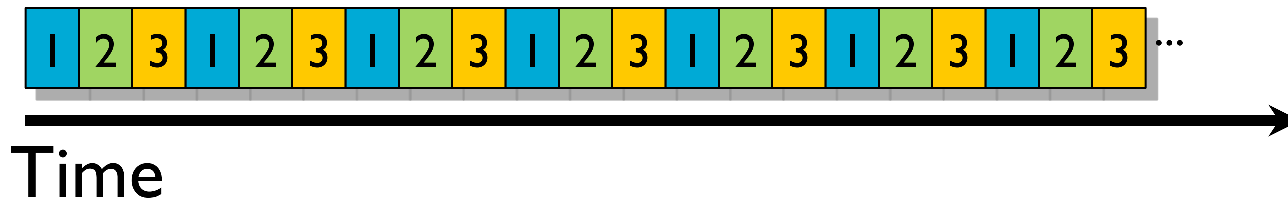
Representative algorithms

- Round-robin
- Priority scheduling

Round-robin

One of the oldest, simplest scheduling algorithms

Select process/thread from ready queue in a round-robin fashion (i.e., take turns)



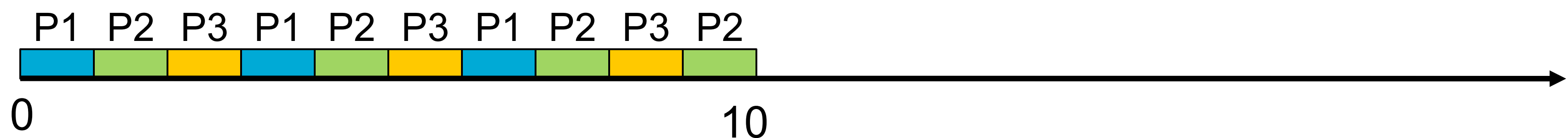
Problems

- Might want some jobs to have greater share
- Context switch overhead

Round-robin: Example

Process	Duration	Order	Arrival Time
P1	3	1	0
P2	4	2	0
P3	3	3	0

Suppose time quantum is 1 unit and P1, P2 & P3 never block



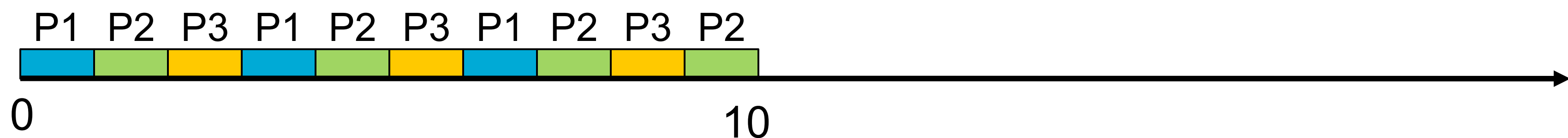
P1 waiting time:
P2 waiting time:
P3 waiting time:

The average waiting time (AWT):

Round-robin: Example

Process	Duration	Order	Arrival Time
P1	3	1	0
P2	4	2	0
P3	3	3	0

Suppose time quantum is 1 unit and P1, P2 & P3 never block



P1 waiting time: 4
P2 waiting time: 6
P3 waiting time: 6

The average waiting time (AWT):
(4+6+6)/3 = 5.33

Round-robin: Summary

Advantages

- Jobs get fair share of CPU
- Shortest jobs finish relatively quickly

Disadvantages

- Larger than optimal average waiting time
  - Example: 10 jobs each requiring 10 time slices
  - RR: All complete after about 100 time slices
  - FCFS performs about 2x better!
- Performance depends on length of time quantum

Choosing the time quantum

How should we choose the time quantum?

Time quantum too large

- FIFO behavior
- Poor response time

Time quantum too small

- Too many context switches (overhead)
- Inefficient CPU utilization

Choosing the time quantum

Heuristic

- 70-80% of jobs block within time-slice

Typical quantum: 1-10 ms

- Large enough that overhead is small percentage
- Small enough to give illusion of concurrency

Question

- If quantum is set to 10 ms, does this mean next process selected for execution will always wait 10 ms before running?

Example: Linux scheduler

Time-sharing scheduler

- assigns a time-slice or quantum to each process

Each processes has a dynamic priority that can be changed by user using the `nice` command:

- very high nice values (+19) provide little CPU time to a process whenever there is any other higher priority load on the system;
- low nice values (-20) deliver most of the CPU to the selected application that requires it (e.g., audio application).

Setting priorities: nice

nice [OPTION] [COMMAND [ARG]...]

- Run COMMAND with an adjusted niceness
- With no COMMAND, print the current niceness.
- Nicenesses range from -20 (most favorable scheduling) to 19 (least favorable).

Options

- -n, --adjustment=N
  - add integer N to the niceness (default 10)
- --help
  - display this help and exit
- --version
  - output version information and exit

Working with priorities in C

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

Get/set scheduling priority of process, process group, or user

Returns:

- setpriority() returns 0 if there is no error, or -1 if there is
- getpriority() can return the value -1, so it is necessary to clear errno prior to the call, then check it afterwards to determine if a -1 is an error or a legitimate value

Parameters:

- which
  - PRIO_PROCESS, PRIO_PGRP, or PRIO_USER
- who

  A process identifier for PRIO_PROCESS, a process group identifier for PRIO_PGRP, or a user ID for PRIO_USER

Experiment: scheduling in practice

```
typedef struct printer_arg_t {
    int thread_index;
} printer_arg_t;

#define BUF_SIZE     100

void * printer_thread( void *ptr )
{
    /* Create the message we will print out */
    printer_arg_t* arg = (printer_arg_t*) ptr;
    char message[BUF_SIZE];
    int i;
    for (i = 0; i < BUF_SIZE; i++)
        message[i] = ' ';
    sprintf(message + 10 * arg->thread_index, "thread %d\n",
            arg->thread_index);

    /* Print it forever */
    while (1)
        printf("%s", message);
}
```

Experiment: results on linux.ews

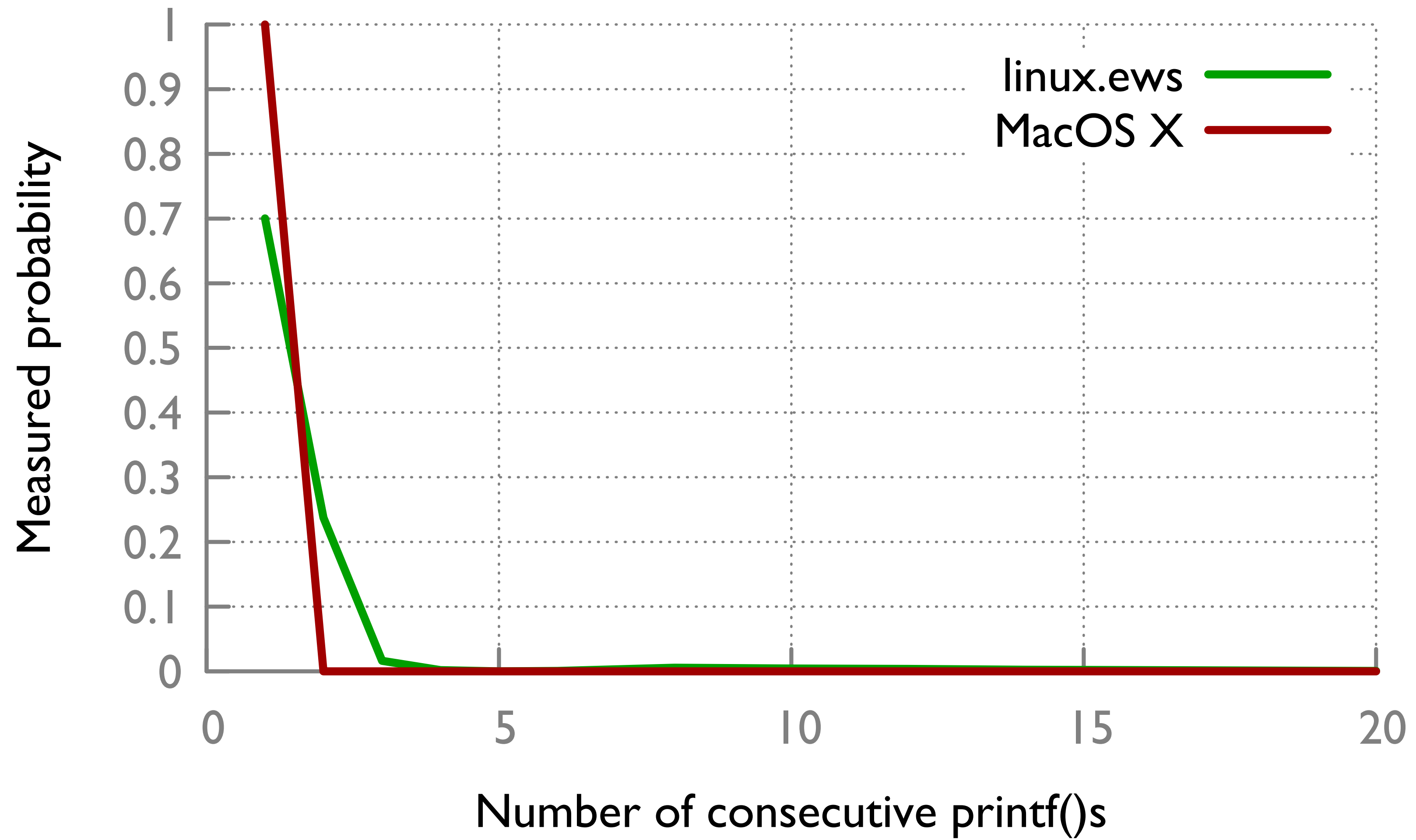
```
          thread 1
          thread 1
          thread 1
          thread 1
          thread 1
          thread 1
          thread 1
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
```

```
thread 0
          thread 1
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
thread 0
...
```

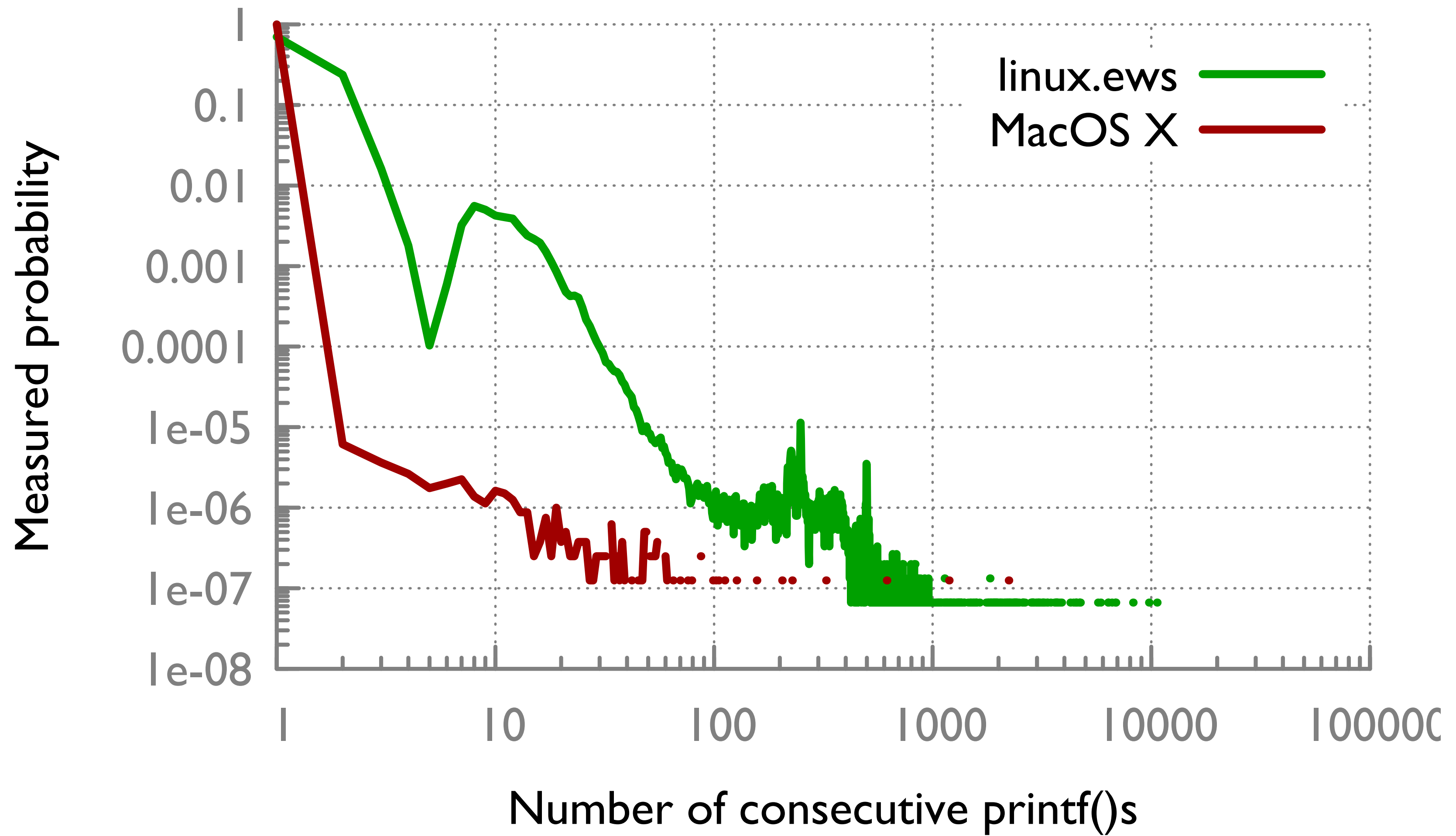
Experiment: results on Mac OS X

```
thread 0
          thread 1
thread 0
          thread 1
thread 0
          thread 1
thread 0
          thread 1
thread 0
          thread 1
thread 0
          thread 1
thread 0
          thread 1
thread 0
          thread 1
thread 0
...
```

Experiment: results



Experiment: results



Take-away point: unpredictability

Scheduling varies across operating systems

Scheduling is non-deterministic even for one OS

- Default (non-real-time) scheduling does not guarantee any fixed length
- Potentially huge variability in work accomplished in one quantum
    - Factor of >10,000 difference in number of consecutive printfs in our experiment!

What if we need some amount of predictability?

Preemptive fixed priority scheduling

Needed for (soft) real-time applications

- Video games, Movie player, interactive

Algorithm

- Each process is assigned a priority
- Scheduler selects highest priority runnable process
- FCFS or Round Robin to break ties

Problems

- May not give the best average waiting time
- But if you need priority scheduling, you care more about deadlines than AWT
- Starvation of lower priority processes

Priority Scheduling: Example

(Lower priority number is preferable)

Process	Duration	Priority	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0

P1 waiting time:
P2 waiting time:
P3 waiting time:
P4 waiting time:

The average waiting time (AWT):

Priority Scheduling: Example

(Lower priority number is preferable)

Process	Duration	Priority	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0

P2 (8) P4 (3) P3 (7) P1 (6)

0 8 11 18 24

P1 waiting time: 18
P2 waiting time: 0
P3 waiting time: 11
P4 waiting time: 8

The average waiting time (AWT):
(0+8+11+18)/4 = 9.25
(worse than SJF's 7)

POSIX real-time scheduling

(available on Linux kernel)

Each process can run with a particular scheduling policy

- SCHED_OTHER: default Linux time-sharing scheduler
- SCHED_FIFO: preemptive, priority-based scheduling
- SCHED_RR: Preemptive, priority-based scheduling with quanta

Scheduling is not clear-cut

Could I have done better? Depends!

- Was some job very high priority?
- Did I know when processes were arriving?
- What's the context switch time?
- What's my objective -- fairness, finish jobs quickly, meet deadlines for certain jobs, ...?
- ...

General-purpose OSes try to perform pretty well for the common case

- Is this good enough to fly an airplane?
- Special purpose (e.g., "hard real-time") scheduling exists
- Linux: "Like all general-purpose operating systems, Linux is designed to maximize average case performance instead of worst case performance. ... if you truly are developing a hard real-time application, consider using hard real-time extensions to Linux ... or use a different operating system"

Scheduling: Issues to remember

Why doesn't scheduling have one easy solution?

What are the pros and cons of each scheduling policy?

How does this matter when you're writing multiprocess/ multithreaded code?

- Can't make assumptions about when your process will be running relative to others!
- May need specialized scheduling for certain applications

Appendix

Posix scheduling interfaces

SCHED_OTHER: **Default Linux time-sharing scheduler**

SCHED_OTHER can only be used at static priority 0. *SCHED_OTHER* is the standard Linux time-sharing scheduler that is intended for all processes that do not require special static priority real-time mechanisms. The process to run is chosen from the static priority 0 list based on a dynamic priority that is determined only inside this list.

The dynamic priority is based on the nice level (set by the **nice** or **setpriority** system call) and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all *SCHED_OTHER* processes.

36

Posix scheduling interfaces

SCHED_FIFO: preemptive, priority-based scheduling.

The available priority range can be identified by calling:
sched_get_priority_min(SCHED_FIFO) ➔ Linux 2.6 kernel: 1
sched_get_priority_max(SCHED_FIFO); ➔ Linux 2.6 kernel: 99

SCHED_FIFO can only be used with static priorities higher than 0, which means that when a *SCHED_FIFO* process becomes runnable, it will always preempt immediately any running *SCHED_OTHER* process. *SCHED_FIFO* is a simple scheduling algorithm without time slicing.

A *SCHED_FIFO* process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, it calls **sched_yield**, or it finishes.

37

Posix scheduling interfaces

SCHED_RR: preemptive, priority-based scheduling with quanta.

The available priority range can be identified by calling:
sched_get_priority_min(SCHED_RR) ➔ Linux 2.6 kernel: 1
sched_get_priority_max(SCHED_RR); ➔ Linux 2.6 kernel: 99

SCHED_RR is a simple enhancement of *SCHED_FIFO*. Everything described above for *SCHED_FIFO* also applies to *SCHED_RR*, except that each process is only allowed to run for a maximum time quantum. If a *SCHED_RR* process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority.

The length of the time quantum can be retrieved by: **sched_rr_get_interval**.

38

Posix scheduling interfaces

Child processes inherit the scheduling algorithm and parameters across a **fork**.

Do not forget!!!!

➔ a non-blocking end-less loop in a process scheduled under *SCHED_FIFO* or *SCHED_RR* will block all processes with lower priority forever

Since *SCHED_FIFO* and *SCHED_RR* processes can preempt other processes forever, only root processes are allowed to activate these policies under Linux.

39

Posix scheduling interfaces

```
#include <sched.h>

#include <sys/types.h>

#include <stdio.h>

main() {

            int                     sched, prio;

            pid_t                   pid;

            struct sched_param  attr;

      printf("\n Scheduling informations: input a PID?\n");
      scanf("%d", &pid);
      sched_getparam(pid, &attr);
      printf("Process %d uses scheduler %d with priority %d \n", pid,
      sched_getscheduler(pid), attr.sched_priority);


      printf("\n Set process sched parameters: PID, scheduler type, priority \n");
      scanf("%d %d %d", &pid, &sched, &prio);


      attr.sched_priority = prio;
      sched_setscheduler(pid, sched, &attr);

}
```

Scheduling: multimedia applications

How should I assign SCHED_RR or SCHED_FIFO priorities?

Multimedia processes usually have periodic computation to run. An optimal policy (called Rate Motonic) assigns process priority as function of its computational period. A timer can be used to awake the process at the beginning of each period

Assume to execute concurrently a 20 frame per second (fps) video application (50msec period) and another 40fps video application (25msec period). You can assign a static priority to each multimedia (real-time) process according to **Rate Monotonic**. Smaller is the period, the higher is the scheduling priority.

See http://en.wikipedia.org/wiki/Rate-monotonic_scheduling

41