# 운영체제 2차 과제

프로세스 및 리눅스 스케줄링의 이해

**고려대학교 운영체제 연구실**

**2019년 5월 7일**

# 과제의 목적

- 프로세스 및 리눅스 스케줄링에 대해 이해한다
- 리눅스 스케줄러 코드를 수정한다
- 내 시스템의 CPU burst 를 직접 측정하고 분석한다

# 프로세스 스케줄링

- **CPU 스케줄링 (프로세스 스케줄링)**
  - 어떻게 프로세스에게 CPU의 사용을 분배할 것인가?
  - 메모리 내 실행이 준비된 프로세들 중에 하나를 선택하여 CPU를 할당

- **스케줄링 디자인**
  - 최대의 CPU 사용률
  - 최소의 응답시간
  - 최소의 대기시간

- **모든 조건을 만족시키는 스케줄러를 만드는 것은 현실적으로 불가능**
- **시스템의 용도에 따른 요구사항이 달라짐**
  - 슈퍼 컴퓨터: CPU 사용률
  - 개인용 컴퓨터: 응답시간

# CFS

- **Completely fair scheduler (CFS)**
  - 리눅스에서 사용되는 스케줄링 방식
  - 여러 프로세스 간의 공평성(fairness)을 보장해주는 스케줄링 방식
  - 성능(performance)과 공평성 간의 균형이 중요
- **각 프로세스의 가상의 CPU 사용 시간에 따라 스케줄링 순서를 결정**
  - CPU 사용 시간이 적은 프로세스에게 기회를 줌
- **예시: CPU 사용 시간에 따른 스케줄링**
  - I/O bound 프로세스들은 CPU bound 프로세스에 비해
    CPU 사용시간이 적으므로, 이를 보상받아 더 높은 CPU 점유 기회를 얻음
- **오랫동안 실행되지 못한 프로세스는 상대적으로 vruntime이 작아지므로, Starvation이 발생하지 않음**

# CFS

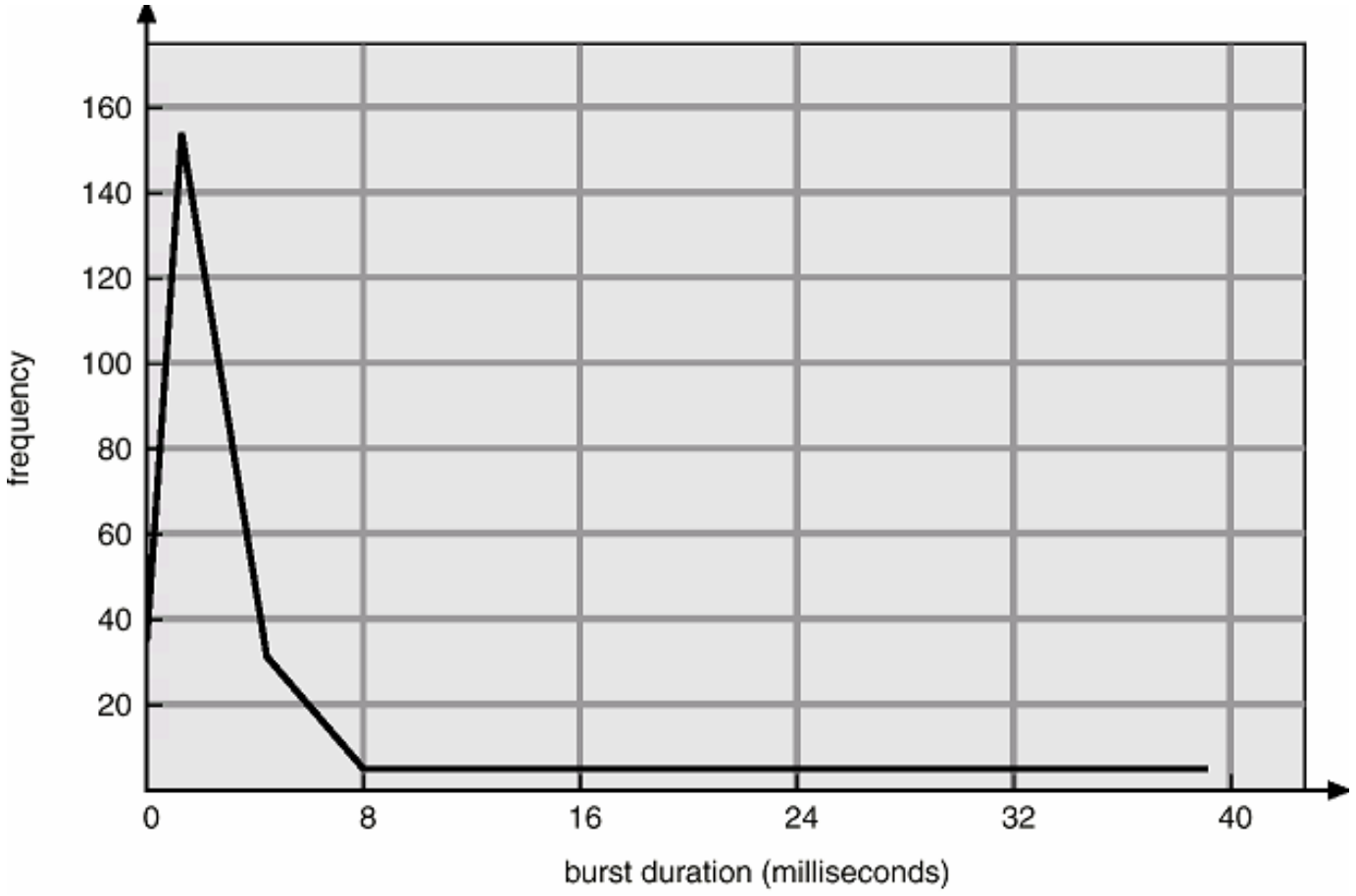
- **CFS의 vruntime**
  - Virtual runtime (vruntime)
  - 가상의 CPU 사용 시간: 가중치(우선순위)가 부여된 CPU 시간
  - 실행 가능한 프로세스들 중, vruntime 이 가장 작은 프로세스를 선택
- **예시**
  - 1) 실제 CPU 사용 시간이 A :100ms, B: 100ms 인 경우
    - 우선 순위가 높은 A의 vruntime: 90ms
    - 우선 순위가 높은 B의 vruntime: 110ms
    - ➔ vruntime이 더 낮은 A를 선택
  - 2) 실제 CPU 사용 시간이 A: 100ms, B: 90ms 인 경우
    - 우선 순위가 높은 A의 vruntime: 90ms
    - 우선 순위가 높은 B의 vruntime: 95ms
    - ➔ vruntime이 더 낮은 A를 선택
- **프로세스의 우선순위를 직접적으로 반영하진 않지만 vruntime 계산에 고려되므로, 스케줄링에 영향을 미침**

# CPU Burst

- **CPU-I/O Burst Cycle**
  - CPU Burst : CPU로 연산을 수행하는 시간.
  - I/O Burst : I/O 처리를 위해 기다리는 시간.
  - 일반적인 프로세스는 두 burst를 번갈아 가며 수행됨
- **프로세스 분류에 따른 CPU Burst의 특징**
  - CPU-bound 프로세스 : 긴 CPU burst
  - I/O-bound 프로세스 : 짧은 CPU burst
- **CPU Burst**
  - 특정 프로세스가 CPU를 점유한 시간
  - CPU 점유 시작 ~ CPU 점유 끝

# Histogram of CPU-burst times

- **CPU burst 빈도수**
  - 서로 다른 프로세스, 시스템에도 불구하고 대체적으로 아래와 같은 경향을 지님
  - 대부분의 프로세스가 상대적으로 작은 CPU burst



# 2차 과제 내용

- CPU burst 측정을 위한 리눅스 스케줄러 수정
- CPU burst 측정 실험 수행
- 프로세스 및 스케줄러에 대한 이해

# 2차 과제 목표

- **2차 과제 목표: CPU burst 측정**
  - 내 시스템에서 프로세스들의 CPU burst는 얼마나 되는지?
  - CPU burst 가 동적인지? 정적인지?

- **CPU burst 측정을 위한 리눅스 스케줄러 수정**
  - 컴파일, 재부팅 과정은 이전 과제와 동일
  - 리눅스 기본 스케줄러에서 할당되는 CPU burst 값을 측정할 수 있도록 스케줄러를 수정
  - CPU burst 값을 로그로 출력 (printk)
  - CPU burst 값은 burst값이 1,000회 바뀔 때 1회 기록

- **다양한 프로세스들의 CPU burst 측정**
  - 웹 브라우저, 음악 재생 프로그램 등등
  - I/O bound, CPU bound 프로세스

# CPU burst 측정을 위한 구현 목표

- **1. 무엇을 출력해야 하는가?**
  - CPU burst

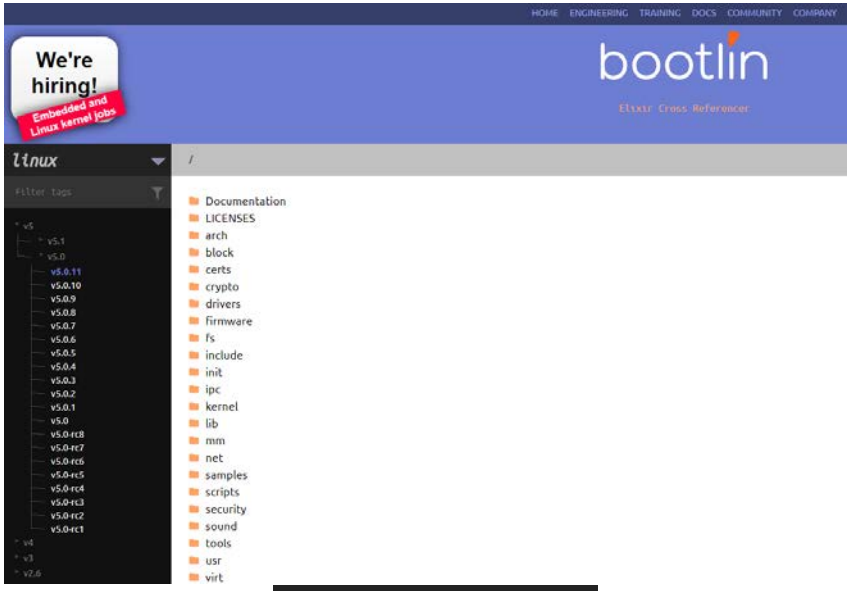
```
[   24.462604] [Pid: 1422], CPUburst: 39672
[   24.462854] [Pid: 1628], CPUburst: 8286
[   24.581598] [Pid: 1437], CPUburst: 51688
[   24.829539] [Pid: 1633], CPUburst: 49082
[   25.384470] [Pid: 1437], CPUburst: 41590
[   25.972033] [Pid: 1628], CPUburst: 25843
[   27.270503] [Pid: 1422], CPUburst: 32380
[   32.755233] [Pid: 1422], CPUburst: 25264
[   33.250812] [Pid: 1982], CPUburst: 4337
```

- **2. 어떻게 샘플링을 해야하는가?**
  - 프로세스마다 1,000회 호출시 CPU burst를 출력하도록
    - 만약 샘플링하지 않으면?
    - 1,000회 샘플링 시 측정 상 병목(bottleneck)이 발생하면 샘플링 횟수를 더 증가시켜도 무방함

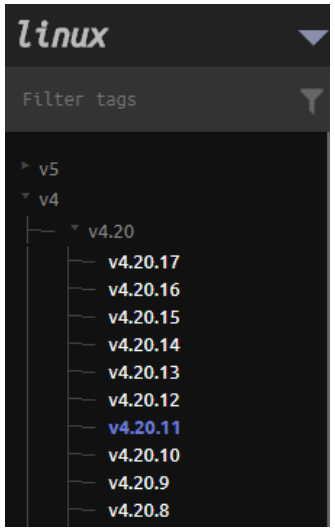
- **sched.h에 있는 특정 자료구조를 이용하여 구현**
  - sched_info_depart() 를 잘 분석할 것

# 리눅스 소스 코드 분석 방법

- **샘플링을 하기 위해 프로세스별 식별 필요**
- **프로세스를 어떻게 식별하는가?**
  - 커널 코드를 어떻게 분석해야 하는가?




- **Tip: bootlin**
  - 리눅스 커널 소스 코드를 온라인으로 볼 수 있는 사이트
  - 함수 및 구조체 검색이 용이함
  - https://elixir.bootlin.com/
  - 다양한 버전의 리눅스 소스코드를 볼 수 있음
    - v4.20.11



# sched_info_depart()

- **목표: CPU burst 출력**
- **프로세스가 CPU 점유를 마쳤을 때 호출되는 함수**
  - 파일: kernel/sched/stats.h

```
static inline void sched_info_depart(struct rq *rq, struct task_struct *t)
{
        unsigned long long delta = rq_clock(rq) - t->sched_info.last_arrival;
```

- **파라미터**
  - struct rq *rq: Run queue
  - task_struct *t: CPU 점유를 마친 프로세스의 PCB 역할을 하는 구조체
    - 리눅스에서는 개념적으로 프로세스 = 쓰레드 = 태스크
- **코드**
  - rq_clock(rq): 현재 시간(ns)
  - t->sched_info.last_arrival: 프로세스가 CPU 점유를 시작했을 때의 시간(ns)

# task_struct 구조체

- **목표: 프로세스별 샘플링**
  - 프로세스(태스크)를 나타내는 구조체를 이용하여 샘플링 구현
- **task_struct 구조체를 분석해야 함**
- **bootlin 을 이용한 task_struct 분석**
  - sched_info_depart() 함수에서 task_struct 클릭
  - Defined 에서 include/linux/sched.h 클릭

/ kernel / sched / stats.h

```
205  * Also, if the process is still in the TASK_RUNNING state, call
206  * sched_info_queued() to mark that it has now again started waiting on
207  * the runqueue.
208  */
209 static inline void sched_info_depart(struct rq *rq, struct task_struct *t)
210 {
211         unsigned long long delta = rq_clock(rq) - t->sched_info.last_arrival;
212
213         rq_sched_info_depart(rq, delta);
214
215         if (t->state == TASK_RUNNING)
216                 sched_info_queued(rq, t);
217 }
218
219 /*
220  * Called when tasks are switched involuntarily due, typically, to expiring
221  * their time slice.  (This may also be called when switching to or from
222  * the idle task.)  We are only called when prev != next.
223  */
224 static inline void
225 __sched_info_switch(struct rq *rq, struct task_struct *prev, struct task_struct *next)
226 {
```

Defined in 2 files:

include/linux/sched.h, line 590 *(as a struct)*

tools/include/linux/lockdep.h, line 26 *(as a struct)*

Referenced in 1245 files:

arch/alpha/include/asm/elf.h, *3 times*

arch/alpha/include/asm/machvec.h, *2 times*

# task_struct 구조체

- **task_struct 구조체에 태스크에 대한 여러가지 정보가 정의되어 있음**
  - 태스크의 상태
    - state
  - 태스크의 식별자
    - pid
- **task_struct 구조체를 이용하여 샘플링을 할 수 있음**

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        /* -1 unrunnable, 0 runnable, >0 stopped: */
        volatile long                   state;

        /*
         * This begins the randomizable portion of task_struct. Only
         * scheduling-critical items should be added above here.
         */
        randomized_struct_fields_start

                                        ...

#endif

        /*
         * May usercopy functions fault on kernel addresses?
         * This is not just a single bit because this can potentially ne
         */
        unsigned int                    kernel_uaccess_faults_ok;

        unsigned long                   atomic_flags; /* Flags requiring

        struct restart_block            restart_block;

        pid_t                           pid;
        pid_t                           tgid;
```

# 구현 내용 요약

- **커널 소스 분석을 통해 구현에 필요한 부분이 어디인 지 파악**
  - Abstraction에 대한 이해, 직관과 경험을 이용해서 찾을 것
- **CPU burst 출력 및 프로세스별 샘플링**
  - 커널 소스를 분석하여 방법을 찾을 것
- **구현 이후 1차과제와 마찬가지로 수행한 것처럼 커널 컴파일**
  - 컴파일 후, 반드시 재부팅해야 새로운 커널이 적용됨
  - 만약 부팅이 안된다면
    - Grub 을 통해서 이전 부팅 이미지로 백업하고 재시도할 것

# CPU burst 측정 및 분석

- **전체 프로세스의 CPU burst 측정**
  - 약 30분 동안의 측정 결과를 분석할 것
  - 기본 커널 쓰레드 이외 **별도의 프로세스를 반드시 실행할 것**
  - 방법
    - 터미널에서 dmesg 명령어 실행
      - 커널 로그를 출력해주는 명령어 (printk를 출력해줌)
      - **출력 되는 시간 단위는 ns**
    - dmesg > log.txt
      - 현재 폴더에서 log.txt 파일에 dmesg 결과를 출력
    - gedit log.txt 로 결과 보기

- **실험결과에 대한 분석 (그래프로 도식화)**
  - 위의 측정 결과를 엑셀 등의 프로그램으로 정리하여 그래프로 그릴 것

# 과제 유의사항

- **제출 마감일**
  - 2019. 06. 04. (화) 오후 11:59
  - 보고서 오프라인 제출: 2019. 06. 05. (수) 오후 05:00
- **Free-day를 사용한 경우 오프라인 보고서는 다음 날 오후 5시까지 제출**
- **최종 제출 마감일 (Free-day 및 출력물 포함)**
  - 온라인 제출 마감: 2019. 06. 09. (일) 오후 11:59
  - 보고서 오프라인 제출 마감: 2019. 06. 10. (월) 오후 05:00
- **구현과 관련된 질문은 받지 않음 (2차과제는 코드 분석 및 구현이 과제의 범위)**
  - 대부분의 질문은 웹 검색을 통해 해결 가능
    - 1차적으로 온라인에서 해결책을 찾아보고 해결할 수 없는 상황에 질문하는 것이 바람직
  - 모든 질문은 “반드시” Facebook page에 먼저 게시할 것
- **질문 사항**
  - Email: osta@os.korea.ac.kr
  - 연구실 방문 : 우정관 308호
    연구실 방문 이전에 반드시 이메일로 문의