학과: 컴퓨터학과
학번: 2022320072
이름: 무함마드 하피주딘 아스라프
제출일: 2025 년 5 월 28 일
과제 2 Freeday 사용 일수: 0 일

1. 개발환경

Host OS 및 개발 도구

- Ubuntu 18.04.02 (64bit)
- GCC 7.5.0
- 터미널: bash

커널 환경

- Linux kernel 4.20.11 (가상머신 환경)
- kernel source 경로: /usr/src/linux-4.20.11
- 커널 빌드 옵션: CONFIG_KU_CPU=y

2. CPU 스케줄링 정책 설명

First Come First Serve (FCFS):
FCFS 방식에서는 스케줄러가 가장 오래 기다린 프로세스를 선택하여 완료될 때까지 실행합니다(비선점형). 이후의 프로세스들은 도착한 순서대로 엄격하게 실행됩니다. 이 방식은 단순하지만, 하나의 긴 작업이 다른 모든 작업을 지연시키는 호송 효과(convoy effect)로 인해 문제가 발생할 수 있습니다. 긴 작업 때문에 긴급하거나 짧은 작업조차도 대기해야 하기 때문입니다.

Shortest Remaining Time First (SRTF):
SRTF 는 선점형 스케줄러로, 항상 남은 실행 시간이 가장 짧은 프로세스를 실행합니다. 만약 새로운 프로세스가 도착했을 때, 그 프로세스의 남은 시간이 현재 실행 중인 프로세스보다 짧다면, 운영체제는 즉시 컨텍스트 스위치(context switch)를 수행합니다. SRTF 는 짧은 작업들의 평균 대기 시간을 최소화할 수 있지만, 짧은 작업들이 계속 도착할 경우 긴 작업이 기아(starvation) 상태에 빠질 수 있다는 단점이 있습니다.

Round Robin (RR):
Round Robin 방식에서는 스케줄러가 준비된 각 프로세스에 고정된 시간 할당량을 부여합니다. 프로세스가 자신의 할당 시간을 모두 사용하면, 해당 프로세스는 선점(preemption)되어 준비 큐의 끝으로 이동하고, 다음 프로세스가 실행됩니다. RR 스케줄링은 어떤 프로세스도 최대 (n – 1)개의 시간 할당량 이상 대기하지 않도록 하여, 응답 시간(response time)에 상한을 두고 공정성(fairness)을 보장합니다. 그러나 많은 프로세스가 동시에 도착하는 경우, 우선순위가 높은 작업이나 짧은 작업이라도 한 주기(n 개의 시간

할당량)를 모두 기다려야 할 수 있으며, 컨텍스트 스위치(context switch)가 자주 발생해 긴 작업의 총 처리 시간(turnaround time)이 증가할 수 있는 단점도 있습니다.

Priority:
Preemptive Priority Scheduling 에서는 각 프로세스에 우선순위 값이 부여되며, 스케줄러는 항상 준비 상태에 있는 프로세스 중 가장 높은 우선순위를 가진 프로세스를 실행합니다. 새로운 프로세스가 도착했을 때, 그 우선순위가 현재 실행 중인 프로세스보다 높다면, 운영체제는 즉시 컨텍스트 스위치(context switch)를 수행합니다. 이 방식은 긴급하거나 중요한 작업을 우선 처리하는 데 유리하지만, 낮은 우선순위의 프로세스는 높은 우선순위 작업이 계속 도착할 경우 기아(starvation) 상태에 빠질 수 있습니다. 이러한 문제를 방지하기 위해, 대기 시간이 길어질수록 프로세스의 우선순위를 점차 높여주는 에이징(aging) 기법이 사용됩니다.

3. 구현 내용

전체 구조

1) 사용자 공간 드라이버 (p.c)

- 다양한 도착 지연 시간, CPU 요구량, (우선순위 정책의 경우) 사용자가 지정한 우선순위 수준을 가진 세 개의 워커 프로세스(A, B, C)를 생성합니다.
- 0.1 초 간격의 "쿼텀(quanta)" 루프를 실행하며, 각 프로세스는 ku_cpu(name, rem_quanta, priority)라는 사용자 정의 시스템 콜을 호출합니다.
- 다음 두 가지 지표를 벽 시계 시간(wall-clock time)으로 측정합니다.
    - **응답 시간**: 프로세스가 도착한 시점부터 최초로 CPU 분할 할당을 받은 시점까지
    - **대기 시간**: 전체 경과 시간에서 실제 CPU 사용 시간을 뺀 시간

2) 커널 공간 시스템 콜 (ku_cpu)

- 하나의 인터페이스 뒤에 네 가지 스케줄링 정책을 구현합니다.

1. **FCFS (First-Come, First-Served)**: 선입선처리:
   단순한 FIFO 대기열

   즉시 실행할 수 없는 새 요청(process)은 항상 대기열(queue)의 꼬리(tail) 에 추가됩니다.

   ```
   list_add_tail(&entry->list, &queue);
   ```

   현재 실행 중인 프로세스가 CPU 를 해제(jobTime == 0)하면, 대기열의 머리(head) 에서 가장 오래 기다린 노드를 꺼냅니다.

   ```
   entry = list_first_entry(&queue, struct job_node, list);
   list_del(&entry->list);
   ```

   이처럼 list_add_tail 과 list_first_entry/list_del 을 짝지어 사용하면 엄격한 FIFO 서비스 순서를 보장합니다.

한 번 “now”가 정해지면 선점 없음

프로세스가 한 번 now = pid 로 CPU 소유자가 되면, jobTime == 0 을 호출해 종료를 알릴 때까지 절대 중간에 빼앗기지 않습니다.

나중에 어떤 프로세스가 더 짧은 작업량을 요청해도, 현재 프로세스가 명시적으로 끝날 때까지 CPU 를 강제로 빼앗지 않습니다.

2. **SRTF (Shortest Remaining Time First)**: 최단 잔여 시간 우선(선점형)
프로세스별 남은 시간 추적 및 선택

   큐에 들어간 각 job_node 와 현재 실행 중인 프로세스에는 rem_time 또는 now_rem 필드로 남은 퀀텀 수를 저장합니다.

   로세스 완료 시 (jobTime == 0), 큐를 전체 순회하여 rem_time 이 가장 작은 노드를 골라냅니다:

```
best = list_first_entry(&queue, struct job_node, list);
list_for_each_entry(entry, &queue, list) {
    if (entry->rem_time < best->rem_time)
        best = entry;
}
list_del(&best->list);
```

   이렇게 선택된 best 노드를 다음 실행 대상으로 삼아, 남은 시간이 가장 짧은 작업을 우선 처리합니다.

   더 짧은 작업의 즉시 선점

   현재 실행 중이 아닌 프로세스가 ku_cpu(name, jobTime)를 호출하거나 남은 시간이 갱신되면, 항상 새 jobTime 과 now_rem 을 비교합니다.

   jobTime < now_rem 이면 즉시 선점(preempt):

   1. 기존 실행 중이던 프로세스를 새로운 job_node 로 만들어 큐 꼬리에 삽입(남은 시간으로 복원).
   2. 호출한 프로세스를 now/now_name/now_rem 으로 승격.
   3. 호출자의 원래 큐 노드를 삭제.
   4. "Working: <이름>" 로그 출력 후 승인(0 반환).

   그렇지 않으면 "Working Denied: <이름>" 로그만 남기고 현재 프로세스 계속 실행(1 반환).

3. **RR (Round Robin)**: 고정 쿼텀(1 초) 기반 라운드 로빈
고정 시간 할당(타임 슬라이스) 선점(quantum_rem)

   초기화: 프로세스가 처음 CPU 를 가져올 때(현재 CPU 가 유휴 상태(now_pid == IDLE_PID)일 때 혹은 턴오버 직후)

```
quantum_rem  = QUANTUM_SLICES;
```

```
#define QUANTUM_SLICES 10
```

슬라이스 회계: 실행 중인 프로세스(pid == now_pid && jobTime > 0)가 다시 호출될 때마다

1. "Working: <now_name>" 출력
2. quantum_rem—
3. quantum_rem > 0 이거나 대기열이 비어 있으면 그대로 계속 실행 (잠금 해제 후 반환)

턴오버: quantum_rem == 0 이고 대기열에 대기 중인 프로세스가 있으면

1. "Turn Over ----> <now_name>" 출력
2. 현재 프로세스를 큐 꼬리에 재삽입 (업데이트된 now_rem 포함)
3. 큐의 머리 노드를 꺼내 now_pid/now_name/now_rem 으로 설정
4. quantum_rem = QUANTUM_SLICES 로 재설정
5. "Working: <새 now_name>" 출력 후 반환 → 다음 프로세스에 CPU 할당

4. **Priority**: 선점형 우선순위 스케줄링(작은 정수값일수록 높은 우선순위)
선점 판단 (Preemption)
현재 실행 중인 프로세스 검사
pid == now_pid 구역에서 대기열(queue)을 순회하며, 자신보다 높은 우선순위(더 작은 priority 값)의 프로세스가 있는지 확인합니다:

```
list_for_each_entry(iter, &queue, list) {
    if (iter->priority < now_prio) {
        preempt = true;
        break;
    }
}
```

선점 수행
preempt == true 이면 즉시 현재 프로세스를 대기열에 다시 삽입하고, 대기열의 머리 노드를 꺼내 새 now_pid/now_name/now_prio/now_rem 으로 교체합니다.

1. 현재 프로세스를 kmalloc 하여 우선순위 순서에 맞게 삽입
2. next = list_first_entry(&queue, ...)로 대기열 머리 선택
3. list_del(&next->list) 후 now_* 갱신
4. "Working: <새 프로세스 이름>" 로그

이로써 더 높은 우선순위가 도착하면 즉시 CPU 를 빼앗아 실행하게 됩니다.

우선순위 기반 정렬 대기열

삽입 시 정렬
CPU 를 점유 중이 아닐 때(pid != now_pid)나, 갱신(update) 시에도 대기열에 넣을 때마다 다음과 같이 적절한 위치를 찾아 삽입합니다:

```
list_for_each(pos, &queue) {
    struct job_node *e = list_entry(pos, struct job_node, list);
    if (entry->priority < e->priority) {
        list_add(&entry->list, pos);
        added = true;
        break;
    }
}
if (!added)
    list_add_tail(&entry->list, &queue);
```

priority 값이 작을수록 높은 우선순위이므로, 더 작은 값 앞에 삽입하여 리스트가 항상 우선순위 순으로 정렬되도록 합니다.

종료 시 다음 할당

jobTime == 0 && pid == now_pid 일 때, "Process Finished" 후 대기열이 비어 있지 않으면 list_first_entry 로 가장 높은 우선순위의 노드를 꺼내 실행합니다.

- 시스템 콜 인터페이스는 아래와 같이 정의됩니다:-

  1) FCFS/SRTF/RR:

```
SYSCALL_DEFINE2(ku_cpu,
    char __user *, uname,
    int,           jobTime)
{
```

  2) Priority:

```
SYSCALL_DEFINE3(ku_cpu,
    char __user *, uname,
    int,           jobTime,
    int,           priority)
{
```

  종합 구현

  run 스크립트는 지정된 지연(delay)과 우선순위(priority policy) 태그를 달아 세 개의 워커 프로세스(A, B, C)를 실행합니다.

  콘솔 로그(p.c 의 printf)는 각 프로세스가 종료될 때마다 메시지를 출력하여 올바른 순서로 종료되었음을 보여줍니다.

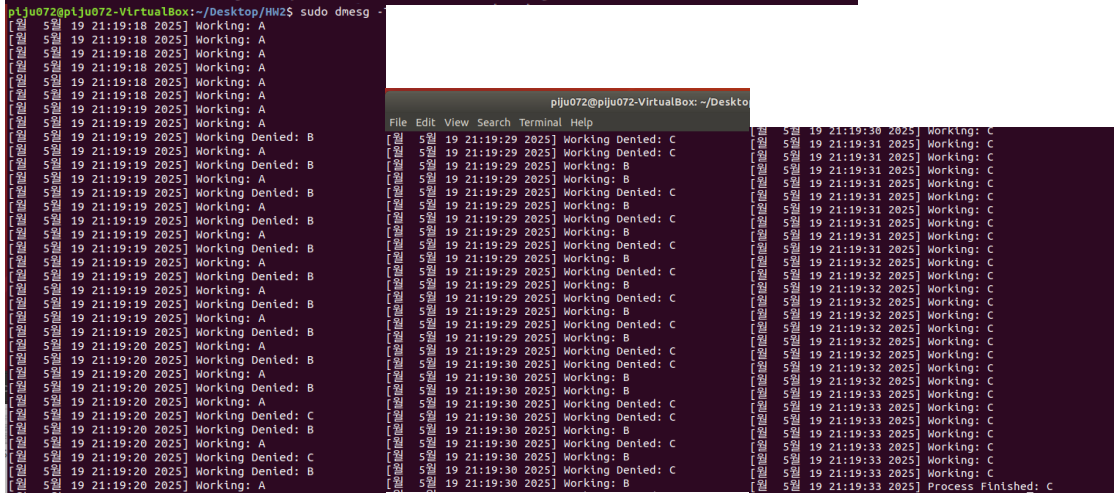
  커널 로그(printk)는 각 쿼텀(slice)별로 할당 허가(grant)·거부(denial)와 최종 종료 시점을 기록하여 스케줄러의 동작을 검증합니다.

  벽 시계 시간(wall-clock time)으로 시작 시각(start), 첫 할당 시각(first grant), 마지막 할당 시각(final grant)을 측정한 뒤 초 단위로 내림(floor) 처리하면, 과제 예제 결과와 정확히 일치함을 확인할 수 있습니다.

4. 실험 결과 및 분석

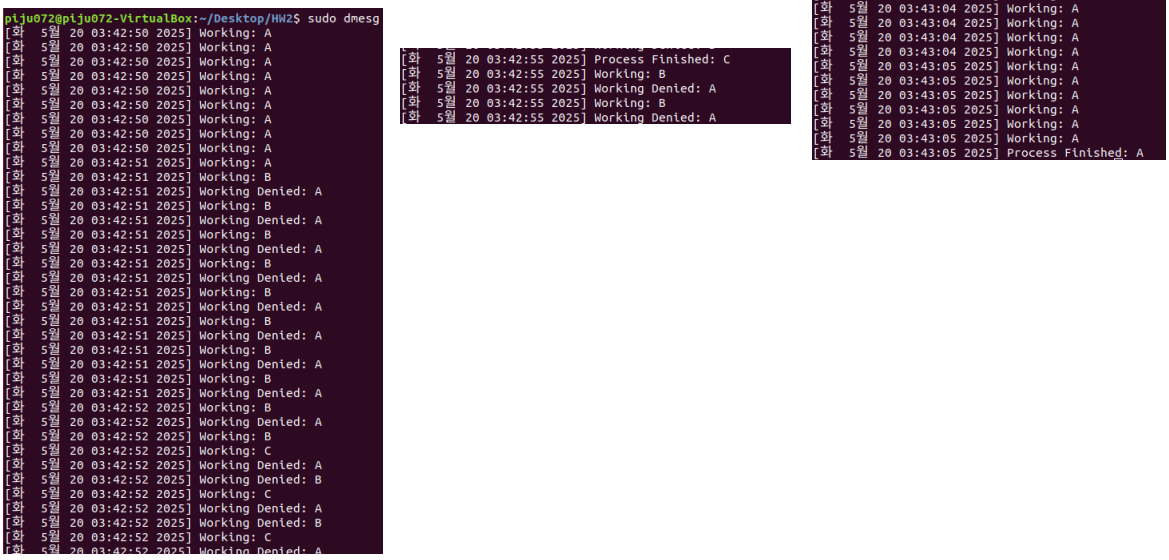
FCFS:

```
piju072@piju072-VirtualBox:~/Desktop/HW2$ ./run

Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process A : Finish! My response time is 0s and My total wait time is 0s.

Process B : Finish! My response time is 6s and My total wait time is 6s.

Process C : Finish! My response time is 10s and My total wait time is 10s.
```



SRTF:

```
piju072@piju072-VirtualBox:~/Desktop/HW2$ ./run

Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process C : Finish! My response time is 0s and My total wait time is 0s.

Process B : Finish! My response time is 0s and My total wait time is 3s.

Process A : Finish! My response time is 0s and My total wait time is 8s.
```



RR:

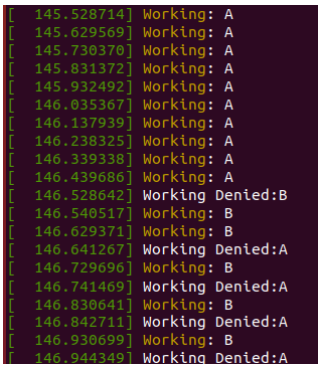
```
piju072@piju072-VirtualBox:~/Desktop/HW2$ ./run

Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process C : Finish! My response time is 1s and My total wait time is 6s.

Process B : Finish! My response time is 0s and My total wait time is 8s.

Process A : Finish! My response time is 0s and My total wait time is 9s.
```

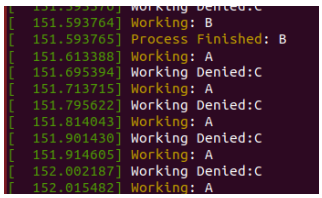
```
[ 9552.395456] Working: A
[ 9552.499451] Working: A
[ 9552.603032] Working: A
[ 9552.703113] Working: A
[ 9552.805439] Working: A
[ 9552.910877] Working: A
[ 9553.011228] Working: A
[ 9553.111894] Working: A
[ 9553.213719] Working: A
[ 9553.316949] Working: A
[ 9553.316950] Turn Over ----> A
[ 9553.397926] Working: B
[ 9553.498182] Working: B
[ 9553.602043] Working: B
[ 9553.704510] Working: B
[ 9553.807987] Working: B
[ 9553.908749] Working: B
[ 9554.012911] Working: B
[ 9554.114645] Working: B
[ 9554.215862] Working: B
[ 9554.318110] Working: B
[ 9554.318111] Turn Over ----> B
[ 9554.350581] Working: A
[ 9554.450704] Working: A
[ 9554.555075] Working: A
[ 9554.655808] Working: A
[ 9554.760966] Working: A
[ 9554.863595] Working: A
[ 9554.970389] Working: A
[ 9555.073909] Working: A
[ 9555.174176] Working: A
[ 9555.280170] Working: A
[ 9555.280172] Turn Over ----> A
[ 9555.302572] Working: C
[ 9555.404478] Working: C
[ 9555.506120] Working: C
[ 9555.606590] Working: C
[ 9555.707615] Working: C
[ 9555.811533] Working: C
[ 9555.912294] Working: C
[ 9556.012850] Working: C
[ 9556.113787] Working: C
[ 9556.214506] Working: C
[ 9556.214507] Turn Over ----> C
```

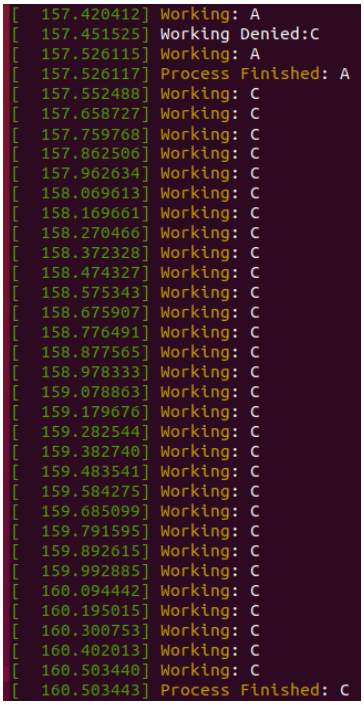
```
[ 9562.840192] Working: B
[ 9562.840193] Turn Over ----> B
[ 9562.895981] Working: A
[ 9562.998114] Working: A
[ 9563.099716] Working: A
[ 9563.200857] Working: A
[ 9563.303747] Working: A
[ 9563.406438] Working: A
[ 9563.508744] Working: A
[ 9563.609517] Working: A
[ 9563.710856] Working: A
[ 9563.812288] Working: A
[ 9563.812290] Turn Over ----> A
[ 9563.859333] Working: B
[ 9563.960733] Working: B
[ 9564.062033] Working: B
[ 9564.165457] Working: B
[ 9564.268685] Working: B
[ 9564.369229] Working: B
[ 9564.470188] Working: B
[ 9564.571621] Working: B
[ 9564.672400] Working: B
[ 9564.772731] Working: B
[ 9564.772731] Turn Over ----> B
[ 9564.833647] Working: A
[ 9564.934124] Working: A
[ 9565.034813] Working: A
[ 9565.137557] Working: A
[ 9565.237847] Working: A
[ 9565.338331] Working: A
[ 9565.438919] Working: A
[ 9565.539354] Working: A
[ 9565.640267] Working: A
[ 9565.742365] Working: A
[ 9565.842948] Working: A
[ 9565.943593] Working: A
[ 9566.044195] Working: A
[ 9566.147086] Working: A
[ 9566.248989] Working: A
[ 9566.349557] Working: A
[ 9566.449747] Working: A
[ 9566.550752] Working: A
[ 9566.651750] Working: A
[ 9566.753021] Process Finished: A
```

Priority:

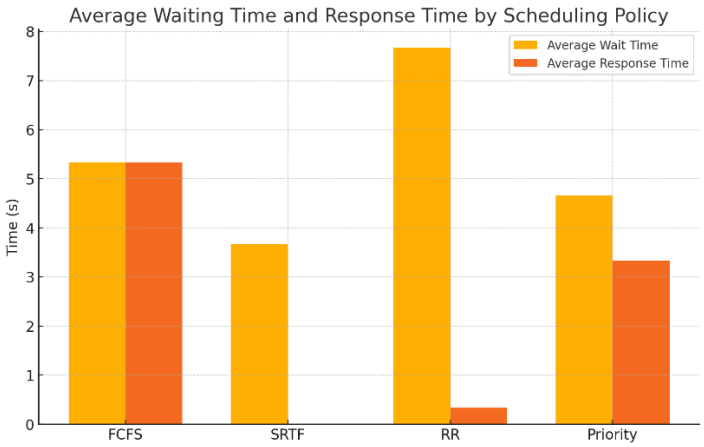
```
piju072@piju072-VirtualBox:~/Desktop/HW2$ ./run
Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process B : Finish! My response time is 0s and My total wait time is 0s.
Process A : Finish! My response time is 0s and My total wait time is 5s.
Process C : Finish! My response time is 10s and My total wait time is 9s.
```







정책 비결:



FCFS (선착순 스케줄링)

장점

구현이 간단하다. 선점(preemption)이나 복잡한 관리 오버헤드가 없다. "먼저 도착한 순서대로 처리"라는 의미에서 공정하다.

단점

컨보이 효과(Convoy effect): 앞에 긴 작업이 있으면 뒤의 모든 작업이 지연된다. 응답 시간과 대기 시간이 모두 높고, 이 둘이 강하게 결합되어 있어 큰 작업 뒤에 도착한 작은 작업이 불리해진다.

적용 사례

작업 길이가 대체로 비슷한 배치(batch) 워크로드. 예측 가능성과 단순성이 응답성보다 더 중요한 시스템(예: 프린트 스풀러).

SRTF (최단 잔여 시간 우선)

장점

평균 대기 시간이 최소이고, 모든 작업의 응답 시간이 0 에 가깝다 짧은 작업은 즉시 시작된다. 길이가 다양한 작업이 섞인 환경에서 전체 처리 시간을 최소화하는 데 뛰어나다.

단점

짧은 작업이 계속 도착하면 긴 작업이 **기아(starvation)** 상태에 빠질 수 있다. 각 작업의 잔여 시간을 알고 있거나 정확히 예측할 수 있어야 한다. 잦은 선점과 리스트 스캔으로 인한 오버헤드가 크다.

적용 사례

작업 길이를 알고 있거나 정확히 예측할 수 있는 배치 또는 대화형 환경. 짧은 작업을 가능한 한 빨리 완료해야 하는 트랜잭션 처리 시스템 등.

RR (Round Robin)

장점

응답 시간 보장: 어떤 작업도 전체 프로세스 사이클 한 번보다 오래 기다리지 않는다. 시간 분할(time-sharing)을 통한 공정성 대화형 시스템에 적합하다.

단점

평균 대기 시간이 모든 정책 중 가장 높다 긴 작업이 여러 퀀텀으로 분할되어 처리된다. 퀀텀마다 문맥 전환(Context switch) 오버헤드가 발생한다.

적용 사례

데스크탑 GUI, 네트워크 서버 등 응답성이 중요한 시분할 운영체제. 많은 대화형 사용자나 프로세스가 공존하는 환경.

Priority (선점형 우선순위 스케줄링)

장점

높은 우선순위 작업은 즉시 실행된다. 우선순위 기반 워크로드에서 FCFS 나 RR 보다 중간 수준의 대기 시간을 보인다.

단점

높은 우선순위 작업이 계속 도착하면 낮은 우선순위 작업이 **기아**에 빠질 수 있다. 우선순위 할당(및 필요 시 에이징)을 신중히 조정해야 한다.

적용 사례

디바이스 처리, 멀티미디어 등 특정 작업이 다른 작업을 선점해야 하는 실시간(Soft-real-time) 시스템. 작업에 의미 있는 우선순위를 부여할 수 있는 모든 환경.