CPU virtualization

구성민
sm.koo1989@gmail.com

CPU Virtualization(1/2)

- CPU virtualization
  - 하나의 CPU 또는 소규모 CPU 집합을 무한개의 CPU가 존재하는 것처럼 변환하여 동시에 많은 수의 프로그램을 실행시키는 것

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/time.h>
4   #include <assert.h>
5   #include "common.h"
6
7   int
8   main(int argc, char *argv[])
9   {
10      if (argc != 2) {
11          fprintf(stderr, "usage: cpu <string>\n");
12          exit(1);
13      }
14      char *str = argv[1];
15      while (1) {
16          Spin(1);
17          printf("%s\n", str);
18      }
19      return 0;
20  }
```

Figure 2.1: **Simple Example: Code That Loops and Prints (cpu . c)**

CPU Virtualization(2/2)

- 독립 수행

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

- 병렬 수행

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Memory Virtualization(1/2)

- Memory
  - Array of bytes

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);        // a4
18      }
19      return 0;
20  }
```

Figure 2.3: **A Program that Accesses Memory (mem.c)**

Memory Virtualization(2/2)

- 독립 수행

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

- 병렬 수행

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Concurrency(1/2)

- 병행성
  - 프로그램이 한 번에 많은 일을 동시에 하려고 할 때 발생하는 문제들

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "common.h"
4
5   volatile int counter = 0;
6   int loops;
7
8   void *worker(void *arg) {
9       int i;
10      for (i = 0; i < loops; i++) {
11          counter++;
12      }
13      return NULL;
14  }
15
16  int
17  main(int argc, char *argv[])
18  {
19      if (argc != 2) {
20          fprintf(stderr, "usage: threads <value>\n");
21          exit(1);
22      }
23      loops = atoi(argv[1]);
24      pthread_t p1, p2;
25      printf("Initial value : %d\n", counter);
26
27      Pthread_create(&p1, NULL, worker, NULL);
28      Pthread_create(&p2, NULL, worker, NULL);
29      Pthread_join(p1, NULL);
30      Pthread_join(p2, NULL);
31      printf("Final value   : %d\n", counter);
32      return 0;
33  }
```

Figure 2.5: A Multi-threaded Program (threads.c)

Concurrency(2/2)

- 증가 연산이 세 개의 명령어(load, increment, store)로 이루어져 있어서 문제 발생
  - Atomic 연산 필요

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298    // what the??
```

Persistence(1/1)

- DRAM과 같은 장치는 volatile이므로 데이터를 저장할 장치 필요
  - Filesystem : 저장 장치를 관리하는 소프트웨어

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <assert.h>
4   #include <fcntl.h>
5   #include <sys/types.h>
6
7   int
8   main(int argc, char *argv[])
9   {
10      int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11      assert(fd > -1);
12      int rc = write(fd, "hello world\n", 13);
13      assert(rc == 13);
14      close(fd);
15      return 0;
16  }
```

Figure 2.6: A Program That Does I/O (io.c)

Process(1/1)

- Program
  - 명령어와 정적 데이터의 묶음
  - Persistent storage에 존재
- Process
  - 실행 중인 프로그램
  - Machine state
    - Memory
    - Register
      - PC(IP), frame pointer, stack pointer
    - Persistent storage
- CPU virtualization
  - Time sharing
  - Context switch
- Mechanism
  - 필요한 기능을 구현하는 방법이나 규칙
- Policy
  - 어떤 결정을 내리기 위한 알고리즘

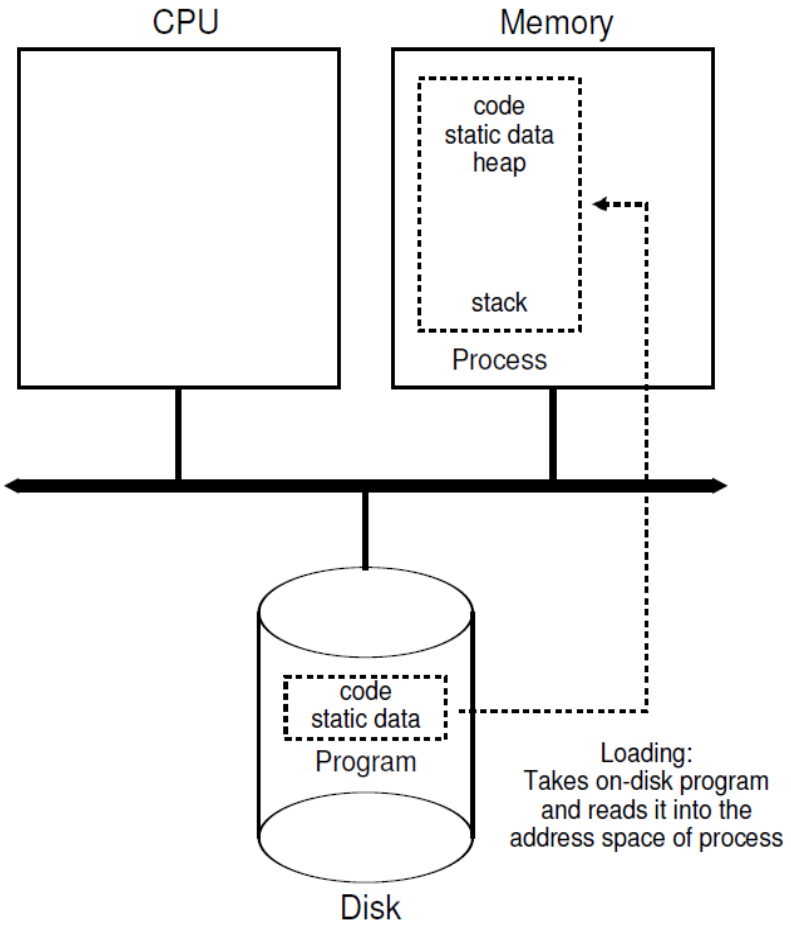


Figure 4.1: **Loading: From Program To Process**

Process API(1/4)

- Create (생성)
  - 새로운 프로세스를 생성하는 인터페이스
  - fork()
- Destroy (제거)
  - 프로세스를 강제로 제거하는 인터페이스
  - kill()
- Wait (대기)
  - 어떤 프로세스의 실행 중지를 기다리는 인터페이스
  - wait(), waitpid()
- Miscellaneous Control (각종 제어)
  - 기타 제어 인터페이스(EX : 일시 정지/해제)
  - signal(), sleep(), lock(), …
- Status (상태)
  - 프로세스 상태 정보를 얻어내는 인터페이스
  - getpid(), getppid(), …

Process API(2/4)

- fork()
  - 자신과 동일한 독립적인 프로세스 생성
  - Return value
    - Parent : 자식의 PID
    - Child : 0
  - 어떤 것이 먼저 실행될지 예측할 수 없음

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {          // fork failed; exit
11         fprintf(stderr, "fork failed\n");
12         exit(1);
13     } else if (rc == 0) { // child (new process)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {               // parent goes down this path (main)
16         printf("hello, I am parent of %d (pid:%d)\n",
17                 rc, (int) getpid());
18     }
19     return 0;
20 }
```

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

Figure 5.1: **Calling `fork()` (`p1.c`)**

Process API(3/4)

- wait()
  - 자식 프로세스 종료 시점까지 자신의 실행을 잠시 중지 시킴
  - 항상 동일한 결과

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int
7  main(int argc, char *argv[])
8  {
9      printf("hello world (pid:%d)\n", (int) getpid());
10     int rc = fork();
11     if (rc < 0) {          // fork failed; exit
12         fprintf(stderr, "fork failed\n");
13         exit(1);
14     } else if (rc == 0) { // child (new process)
15         printf("hello, I am child (pid:%d)\n", (int) getpid());
16     } else {               // parent goes down this path (main)
17         int wc = wait(NULL);
18         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19                rc, wc, (int) getpid());
20     }
21     return 0;
22 }
```

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

Figure 5.2: **Calling `fork()` And `wait()` (`p2.c`)**

Process API(4/4)

- exec()
  - fork()로 생성된 프로세스를 다른 프로그램으로 대체

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     printf("hello world (pid:%d)\n", (int) getpid());
11     int rc = fork();
12     if (rc < 0) {          // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) { // child (new process)
16         printf("hello, I am child (pid:%d)\n", (int) getpid());
17         char *myargs[3];
18         myargs[0] = strdup("wc");   // program: "wc" (word count)
19         myargs[1] = strdup("p3.c"); // argument: file to count
20         myargs[2] = NULL;           // marks end of array
21         execvp(myargs[0], myargs);  // runs word count
22         printf("this shouldn't print out");
23     } else {               // parent goes down this path (main)
24         int wc = wait(NULL);
25         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26                rc, wc, (int) getpid());
27     }
28     return 0;
29 }
```

Figure 5.3: **Calling `fork()`, `wait()`, And `exec()` (p3.c)**

Process State(1/2)

- Running
  - Process가 processor에서 실행 중인 상태
- Ready
  - 프로세스가 실행 할 준비가 된 상태이지만 대기 중인 상태
- Blocked
  - 프로세스가 다른 사건(ex: I/O, wait packet etc..)을 기다리는 동안 중단된 상태

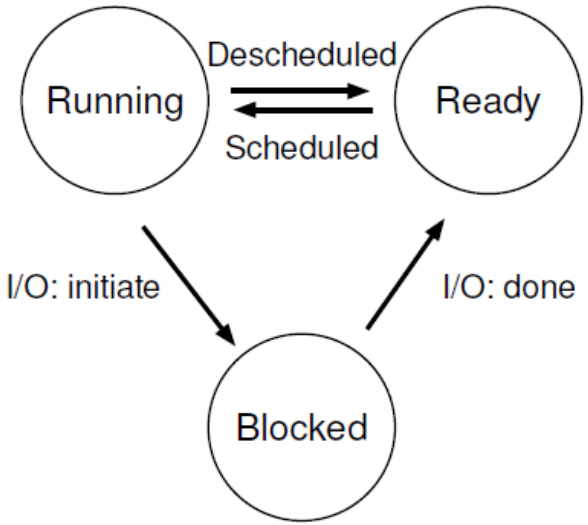


Figure 4.2: **Process: State Transitions**

Process State(2/2)

- State transition example

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

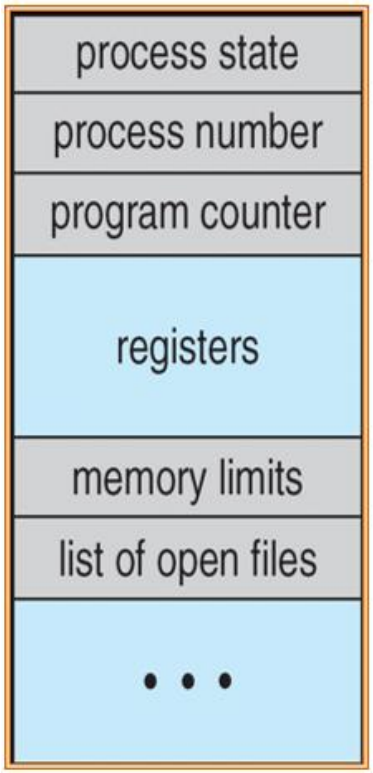
Figure 4.3: **Tracing Process State: CPU Only**

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

Process Data structure(1/1)

- xv6 kernel data structure
  - PCB
    - list로 관리
    - struct proc, task_struct




```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Figure 4.5: The xv6 Proc Structure

Limited Direct Execution(1/2)

- Time sharing
  - CPU virtualization의 key mechanism
  - Issues
    - Performance
      - 어떻게 CPU 가상화 overhead를 최소화 할까?
    - Control
      - 어떻게 CPU 통제를 유지하며 프로세스를 수행시킬까?

Limited Direct Execution(2/2)

- Direct execution
  - 프로그램을 CPU 상에서 직접 실행
  - Problem
    - 어떻게 프로그램이 운영체제가 원하지 않는 일을 하지 않도록 보장할까?
      - I/O, memory 할당 etc…
    - OS가 어떻게 프로그램의 실행을 중단하고 다른 프로세스로 전환시킬 수 있을까?

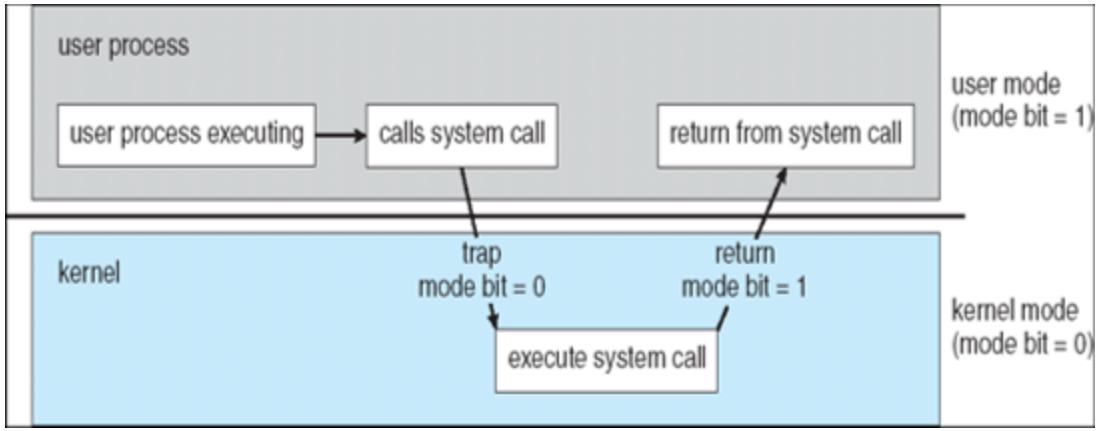
OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute call main()	
	Run main()
	Execute return from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

Problem #1: 제한된 연산(1/2)

- 제한된 연산
  - Privileged mode에서만 수행 가능한 operations
    - CPU, memory등의 system resource 추가 할당
    - I/O request
  - System Call
    - fork(), malloc(), nice(), open(), read(), write, …

- User mode vs Kernel mode
  - User mode: privileged operation 수행 시 exception 발생 후 종료
  - Kernel mode: privileged operation 수행 가능
  - Mode switch : trap instruction을 통해 수행



Problem #1: 제한된 연산(2/2)

- Trap table(interrupt vector table)
  - Trap handler들의 set으로 구성
  - System call handler, div_by_zero handler, segment fault handler, page fault handler, hardware interrupt handler(disk, keyboard, timer etc…)
  - 부팅 시 초기화 됨

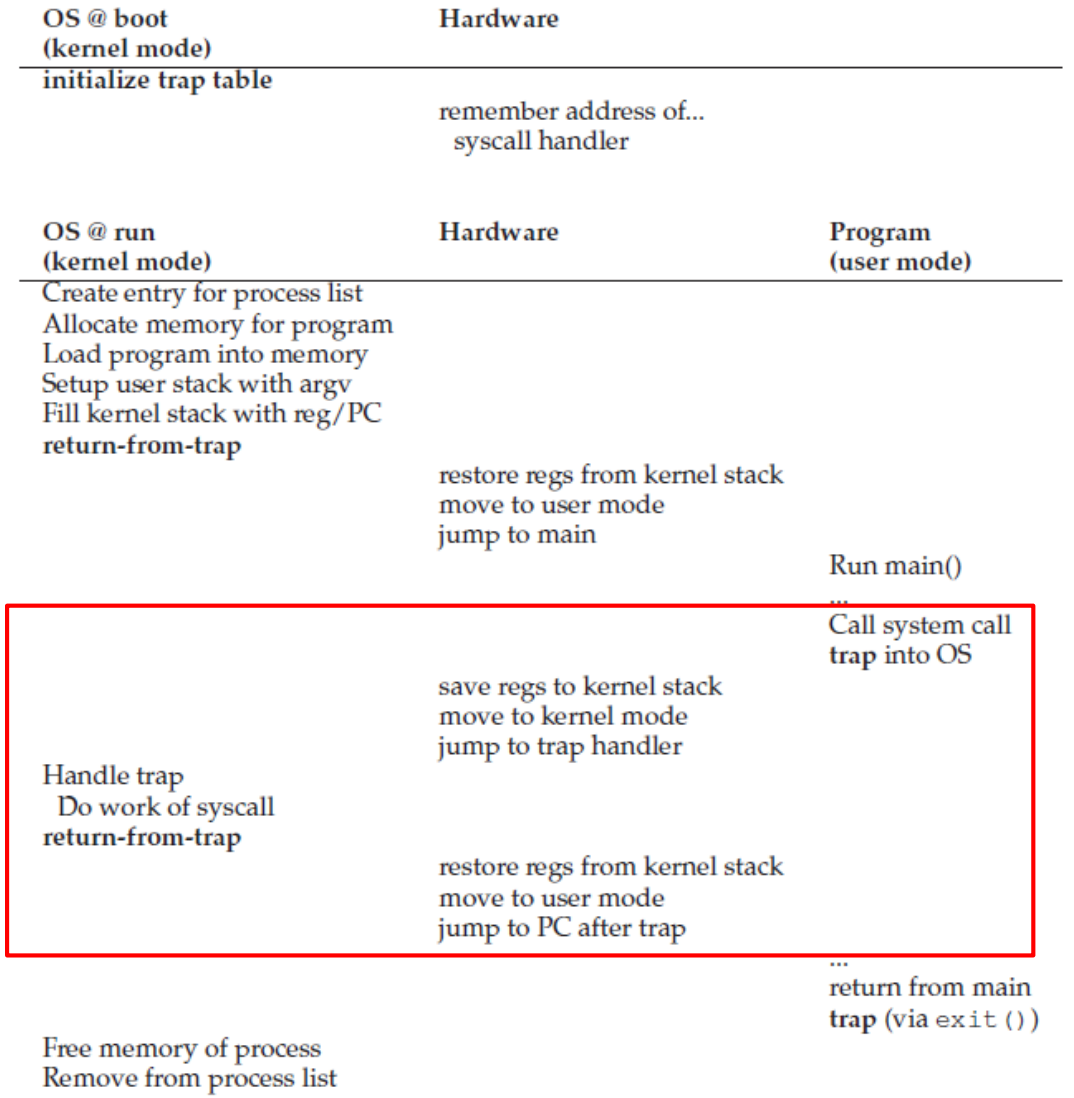


Figure 6.2: Limited Direct Execution Protocol

Problem #2: 프로세스간 전환(1/3)

- Cooperative
  - System call 사용 ➔ OS가 제어권 획득 ➔ scheduling
  - Exception 발생(page fault, divide by zero) ➔ OS가 제어권 획득 ➔ 해당 프로세스 종료
  - yield() system call
    - System call이 거의 호출 되지 않는 경우 'yield' system call을 호출 하여 OS에게 제어권을 넘긴다.
  - 프로세스가 무한 루프에 빠지면 재부팅 밖에 답이 없다.

- Non-cooperative
  - Timer interrupt를 이용하여 일정 기간 마다 interrupt를 발생 시켜 제어권을 OS가 제어권을 획득 하도록 한다.

Problem #2: 프로세스간 전환(2/3)

- Context switch view

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) from k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

Problem #2: 프로세스간 전환(3/3)

- Context switch pseudo code

```
1   # void swtch(struct context **old, struct context *new);
2   #
3   # Save current register context in old
4   # and then load register context from new.
5   .globl swtch
6   swtch:
7     # Save old registers
8     movl 4(%esp), %eax  # put old ptr into eax
9     popl 0(%eax)        # save the old IP
10    movl %esp, 4(%eax)  # and stack
11    movl %ebx, 8(%eax)  # and other registers
12    movl %ecx, 12(%eax)
13    movl %edx, 16(%eax)
14    movl %esi, 20(%eax)
15    movl %edi, 24(%eax)
16    movl %ebp, 28(%eax)
17
18    # Load new registers
19    movl 4(%esp), %eax  # put new ptr into eax
20    movl 28(%eax), %ebp # restore other registers
21    movl 24(%eax), %edi
22    movl 20(%eax), %esi
23    movl 16(%eax), %edx
24    movl 12(%eax), %ecx
25    movl 8(%eax), %ebx
26    movl 4(%eax), %esp  # stack is switched here
27    pushl 0(%eax)       # return addr put in place
28    ret                 # finally return into new ctxt
```

Figure 6.4: The xv6 Context Switch Code

Concurrency(1/1)

- Issue
  - System call을 처리하는 도중에 timer interrupt가 발생
  - 인터럽트 처리 도중에 다른 인터럽트가 발생

- Solution
  - Disable interrupt
  - 우선 순위 설정
  - Locking mechanism

Scheduling(1/2)

- Scheduling
  - Processor에서 수행 될 process를 선택하는 작업
- Workload
  - 프로그램이 수행될 때 필요한 resource의 양
- 가정
  - 모든 작업은 같은 시간 동안 실행된다.
  - 모든 작업은 동시에 도착한다.
  - 각 작업은 시작되면 완료될 때까지 실행된다.
  - 모든 작업은 CPU만 사용한다. (I/O를 수행하지 않는다.)
  - 각 작업의 실행 시간은 사전에 알려져 있다.
- 가정이 비현실적이지만, 차차 가정을 완화시킬 예정

Scheduling(2/2)

- Metric (평가 항목)
  - 측정을 위해 사용할 지표 (ex: performance, fairness, …)

- Scheduling metric
  - Turnaround time (반환 시간)
    - $T_{turnarond} = T_{completion} - T_{arrival}$
    - $T_{arrival}$ = 0
  - Response time
    - $T_{response} = T_{firsturn} - T_{arrival}$
  - Fairness
  - Throughput
  - Deadline
  - …

FIFO(1/1)

- First In First Out = FCFS(First Come First Serve)
  - 먼저 도착한 순서대로 수행
  - Example 1
    - Run time: A – 10s, B – 10s, C – 10s

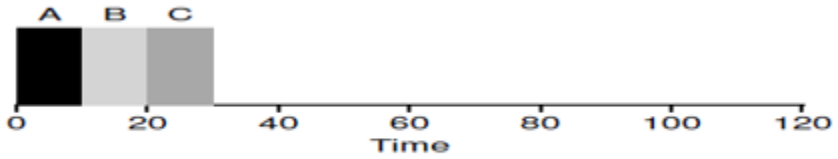


Figure 7.1: FIFO Simple Example

    - Turnaround time: $\frac{10+20+30}{3} = 20$
  - Example 2
    - Run time: A – 100s, B – 10s, C – 10s

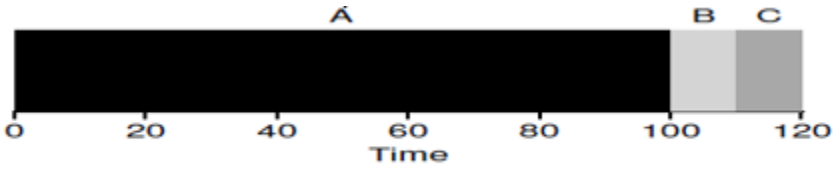


Figure 7.2: Why FIFO Is Not That Great

    - Turnaround time: $\frac{100+110+120}{3} = 110$ (convoy effect)

SJF(1/1)

- Shortest Job First = SPN(Shortest Process Next)
  - 가장 짧은 실행 시간을 가진 작업을 먼저 실행
  - Previous Example
    - Run time: A – 100s, B – 10s, C – 10s

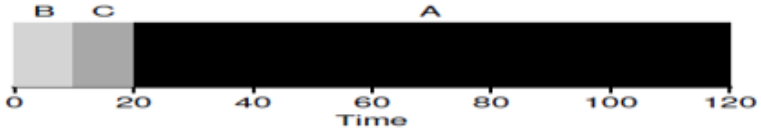


Figure 7.3: SJF Simple Example

    - Turnaround time: $\frac{10+20+120}{3} = 50$
  - Example
    - 동시에 도착하지 않을 경우(A: t = 0, B, C: t = 10)

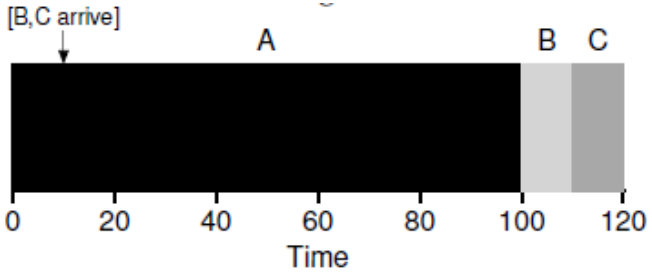


Figure 7.4: SJF With Late Arrivals From B and C

    - Turnaround time: $\frac{10+(110-10)+(120-10)}{3} = 103.33$
    - Convoy effect 발생

STCF(1/1)

- Shortest Time-to-Completion First = SRT(Shortest Remaining-Time next)
  - Preemptive SJF
  - Non-preemptive scheduling
    - 작업이 완료될 때 까지 수행
  - Preemptive scheduling
    - 작업을 중지시키고 다른 작업을 수행 할 수 있음
    - Context switch 사용
  - Example
    - 동시에 도착하지 않을 경우(A: t = 0, B, C: t = 10)

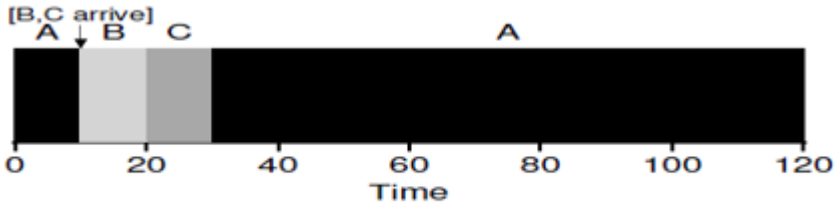


Figure 7.5: STCF Simple Example

- Turnaround time: $\frac{(120-0)+(20-10)+(30-10)}{3} = 50$

Response time(1/1)

- Batch system
  - 작업의 길이를 알고 있고, CPU만 사용하며, 평가 metric이 반환 시간 하나라면, STCF는 좋은 정책

- Time-sharing system
  - 현재 시스템처럼 terminal 작업의 경우 응답시간이 매우 중요
  - Response time
    - $T_{response} = T_{firstrun} - T_{arrival}$
  - Example with SJF

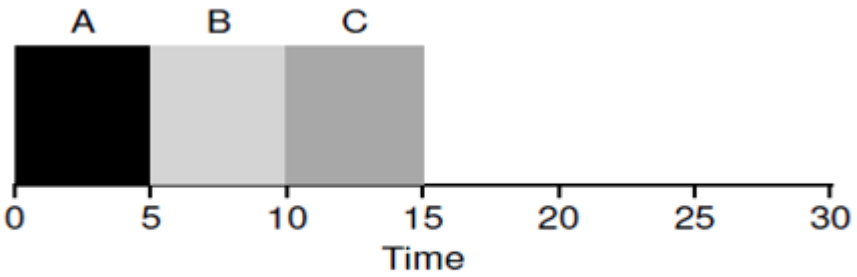


Figure 7.6: **SJF Again (Bad for Response Time)**

  - Response time
    - A: 0s, B: 5s, C: 10s, avg: $\frac{0+5+10}{3} = 5$

RR(1/2)

- Round-Robin
  - Time slice(Scheduling quantum) 동안 실행한 후 run queue의 다음 작업으로 전환
  - 작업이 끝날 때 까지 반복적으로 수행
  - Example
    - Run time: 5s, arrival time: 0s
    - Time slice = 1s

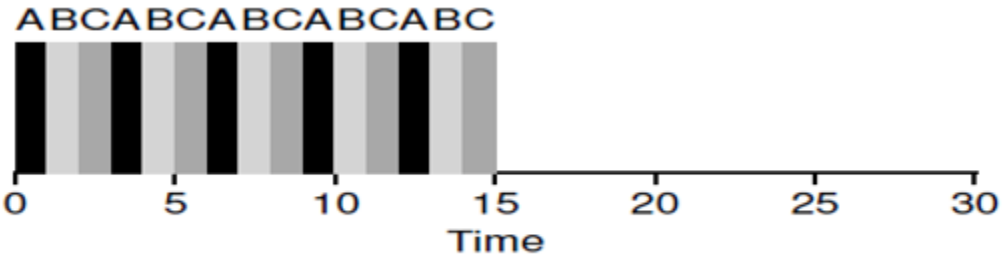
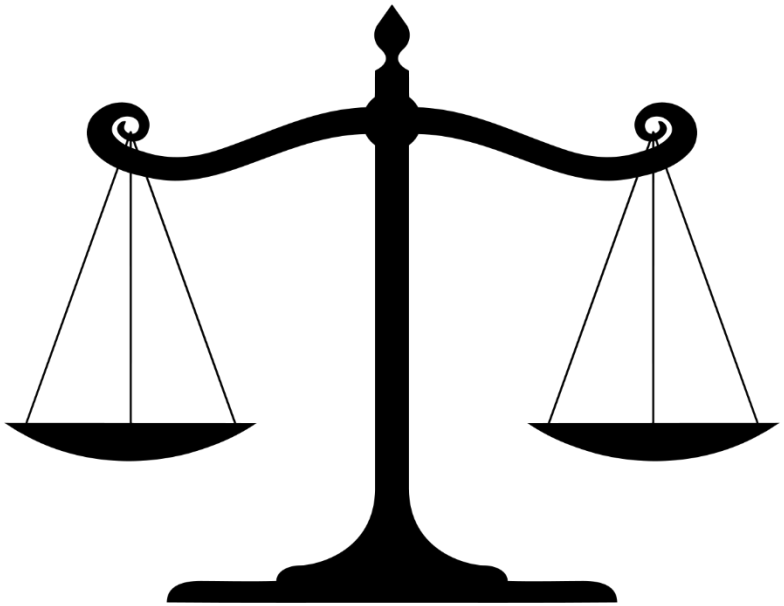


Figure 7.7: Round Robin (Good for Response Time)

    - Average response time
      - $\frac{0+1+2}{3} = 1$
    - Turnaround time
      - $\frac{13+14+15}{3} = 14$

RR(2/2)

- Time quantum의 overhead
  - Response time vs turnaround time
  - Fairness vs performance
  - Small
    - 빠른 응답성, 높은 context switch overhead
  - Large
    - 느린 응답성, 낮은 context switch overhead



I/O(1/1)

- 대부분의 app은 I/O가 발생한다.
  - CPU burst 크기로 독립적인 작업으로 나누고 가장 짧은 작업을 선택
- Example
  - A,B: 50ms CPU 사용
  - A: 10ms 실행 후, 10ms I/O 수행
  - CPU burst
    - A: 10ms, B: 50ms

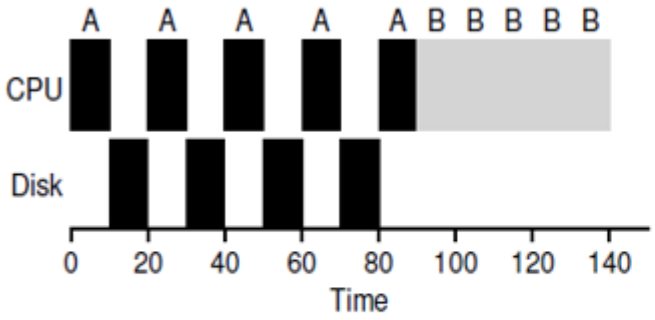


Figure 7.8: **Poor Use of Resources**

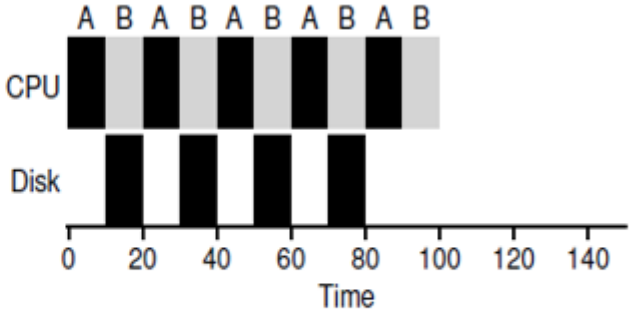


Figure 7.9: **Overlap Allows Better Use of Resources**

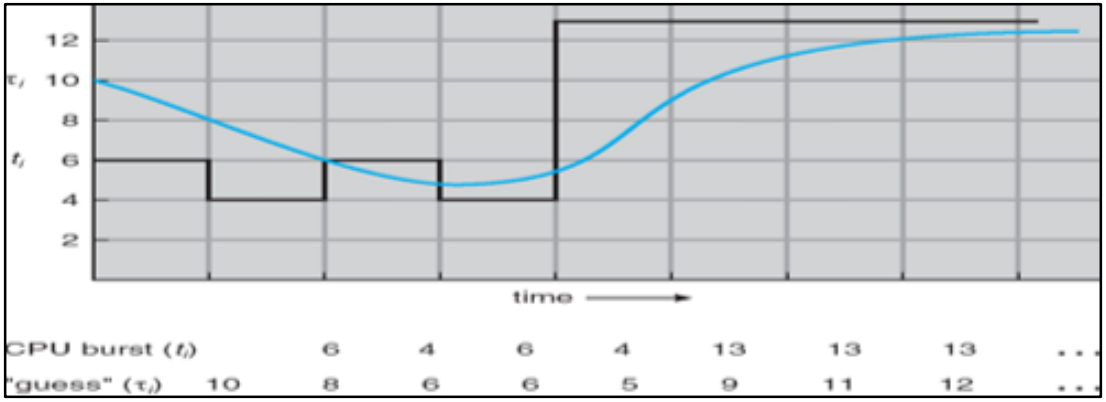
No more Oracle(1/1)

- 실제 OS에서는 작업의 길이를 알 수 없다.
  - CPU 사용 시간은 이전의 CPU 사용시간에 비례

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

where
- τ_{n+1} = predicted value for the next CPU burst
- t_n = actual length of n^{th} CPU burst
- $\alpha, 0 \le \alpha \le 1$

  - $\alpha = 0.5, \tau_0 = 10$



MLFQ(1/8)

- Multi Level Feedback Queue
  - 짧은 작업을 먼저 실행시켜 Turnaround time을 최적화 (SJF, STCF)
  - 대화형 사용자를 위해 Response time 을 최적화 (RR)

- Basic rule
  - **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
  - **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.

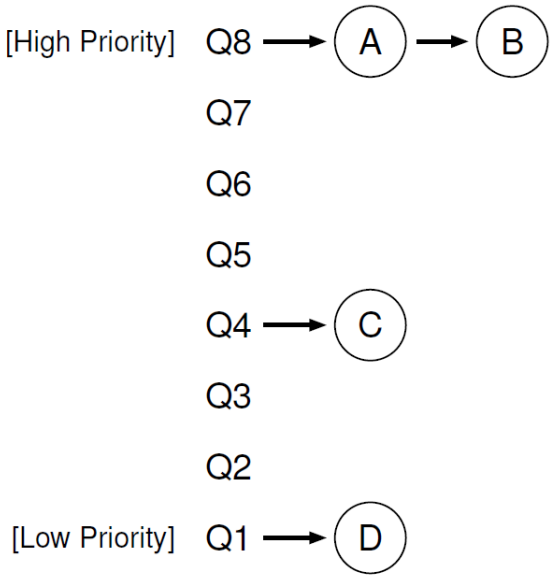


Figure 8.1: **MLFQ Example**

MLFQ(2/8)

- 우선순위의 변경
  - Priority가 높은 process들만 번갈아 수행되고, Priority가 낮은 process들은 영원히 수행되지 못함.
  - 잦은 I/O ➔ 대화형 ➔ 높은 priority ➔ 상위 queue
  - CPU intensive ➔ batch ➔ 낮은 priority ➔ 하위 queue로 이동

- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

MLFQ(3/8)

- Example 1
  - 한 개의 긴 실행 시간을 가진 작업, time quantum = 10ms
- Example 2
  - A: 긴 작업, B: 짧은 작업(20ms)
- Example 3
  - A: 긴 작업, B: I/O intensive(CPU 1ms 이용)
  - Time quantum 이전에 CPU를 반환 하므로 상위 queue에 유지됨(rule 4b)

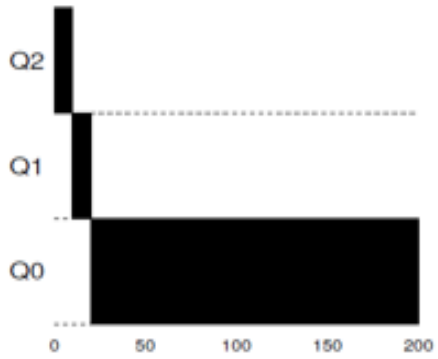


Figure 8.2: **Long-running Job Over Time**

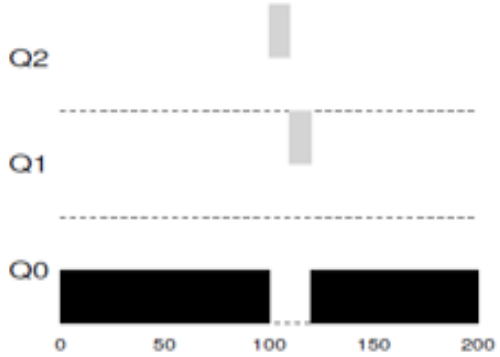


Figure 8.3: **Along Came An Interactive Job**

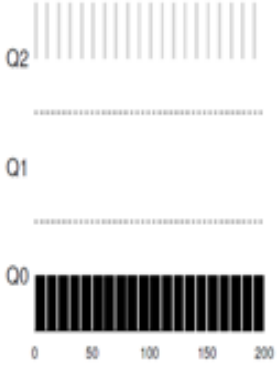


Figure 8.4: A Mixed I/O-intensive and CPU-intensive Workload

MLFQ(4/8)

- MLFQ의 장점
  - 긴 작업들과 짧은 작업들 사이에서 CPU를 잘 공유
  - I/O 중점 대화형 작업을 빨리 수행

- MLFQ의 문제
  - Starvation(기아 상태)
    - 시스템에 많은 대화형 작업들이 존재하면, 대화형 작업들이 모든 CPU 시간을 소모하여 긴 실행 시간을 갖는 작업은 CPU를 할당 받지 못한다.
  - Gaming of scheduler
    - Scheduler를 속여서 지정된 몫보다 더 많은 시간을 할당하도록 하는 프로그램을 작성 할 수 있다.
      - Ex: 99% CPU 사용 후 임의의 I/O 발생 ➔ 상위 queue에 프로세스 유지
  - 프로그램 특성의 변화
    - CPU위주 작업 ➔ 대화형 작업

MLFQ(5/8)

- 우선순위 상향 조정
  - New rule
    - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.
  - Starvation, 프로그램 특성의 변화 문제 해결

- Example
  - 2개의 대화형 작업, 1개의 긴 실행 시간을 갖는 작업
  - 50ms마다 우선순위 상향 조정

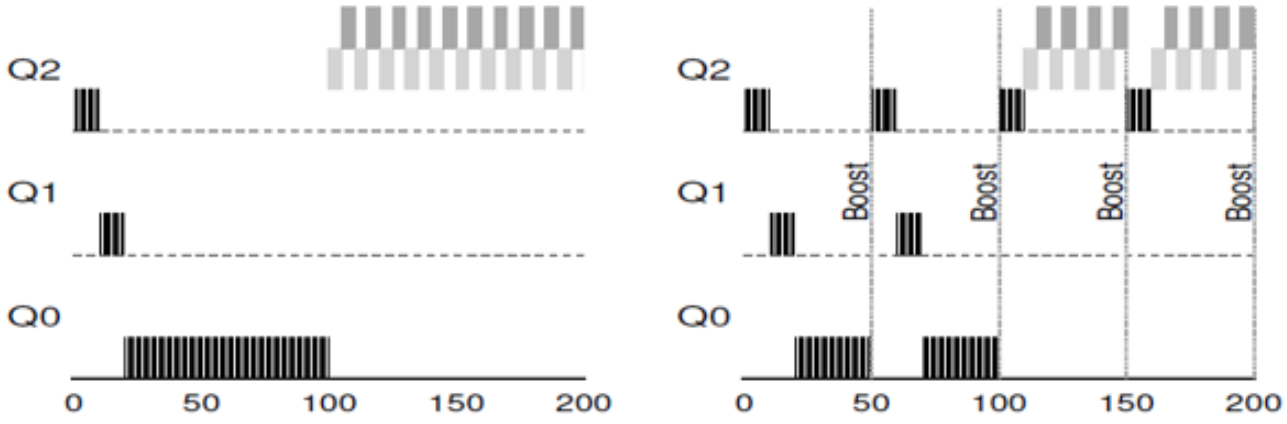


Figure 8.5: **Without (Left) and With (Right) Priority Boost**

MLFQ(6/8)

- Gaming of scheduler을 막는 방법
  - CPU 총 사용 시간을 측정하여 타임 슬라이스에 해당하는 시간을 모두 소진하면 다음 우선순위 queue로 이동
  - Rule 4a + Rule 4b ➔ Rule 4
    - **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Example

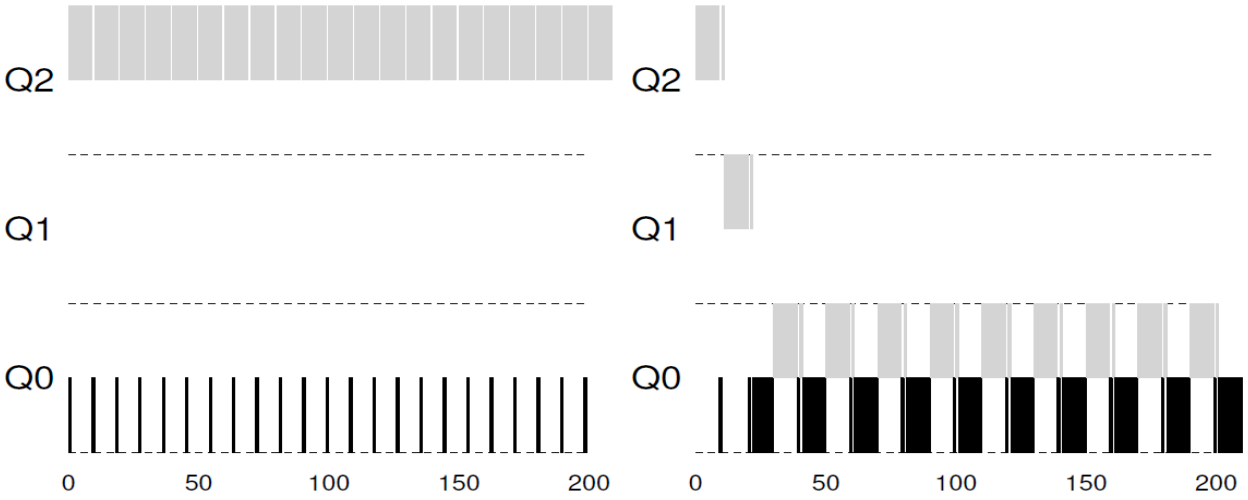


Figure 8.6: **Without (Left) and With (Right) Gaming Tolerance**

MLFQ(7/8)

- Issues
  - Queue의 개수
  - Queue의 time quantum 크기
    - Queue마다 보통 time quantum의 크기가 다르다.(상위 queue일 수록 짧음)
  - 우선 순위 상향의 주기

- 대부분의 OS에서 parameter를 통해 issue가 되는 것들을 setting할 수 있다.

- Example
  - Queue마다 서로 다른 크기의 time quantum

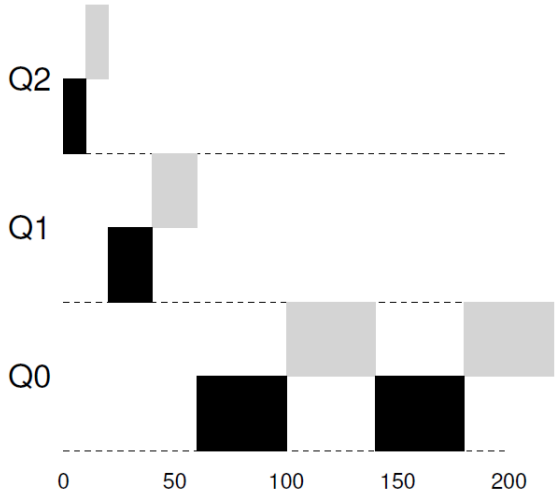


Figure 8.7: **Lower Priority, Longer Quanta**

MLFQ(8/8)

- Summary
  - CPU intensive 작업과 interactive job에게 모두 좋다.

- **Rule 1:** If Priority(A) $>$ Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) $=$ Priority(B), A & B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

Proportional Share(1/5)

- Proportional Share(Fair Share)
  - Turnaround time이나 response time 대신에, scheduler가 각 프로세스에게 CPU의 일정 비율을 보장하는 것이 목적
  - Scheduling algorithms: Lottery, Stride, …

- Basic concept: Tickets (Lottery scheduling)
  - 프로세스가 받아야할 자원의 몫
    - 프로세스가 소유한 티켓/전체 티켓 = 자신의 몫
  - Example
    - Total: 0-99, A: 0-74, B: 75-99

Here is an example output of a lottery scheduler's winning tickets:

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49 49
```

Here is the resulting schedule:

```
A     A  A     A  A  A  A  A  A     A     A  A  A  A  A  A
   B        B                    B     B
```

    - A: 80%, B: 20%
      - 장시간 실행 시 원하는 비율을 달성할 가능성이 높아진다.

Proportional Share(2/5)

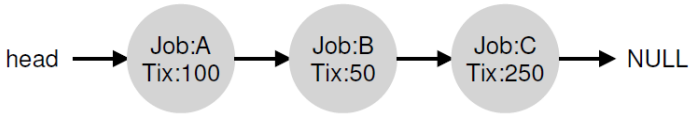
- Ticket mechanism
  - Ticket currency
    - 사용자가 ticket을 자신의 작업에 자신의 화폐 가치로 자유롭게 할당
      - 시스템이 자동적으로 화폐 가치를 변환
    - Example
      - Global ticket: 200(A: 100, B: 100)
      - A: 1000(A1: 500, A2: 500 – 두개의 작업), B: 10 - 1개의 작업

```
User A -> 500 (A's currency) to A1 ->  50 (global currency)
       -> 500 (A's currency) to A2 ->  50 (global currency)
User B ->  10 (B's currency) to B1 -> 100 (global currency)
```

  - Ticket transfer
    - 프로세스는 일시적으로 ticket을 다른 프로세스에게 넘겨줄 수 있다.
    - Client/Server 환경에서 유용
      - Server에게 특정 작업을 요청 시, ticket도 일시적으로 같이 양도
  - Ticket inflation
    - 프로세스는 일시적으로 자신이 소유한 추첨권의 수를 늘리거나 줄일 수 있다.
    - Cooperative environment에서 사용(경쟁 환경에선 의미가 없음)

Proportional Share(3/5)

- Implementation
  - 난수 발생기, 전체 티켓의 수, 프로세스 리스트
  - Counter의 값이 winner의 값을 초과할 때 까지 각 ticket의 개수를 counter에 더함.
  - 값이 초과하게 되면, 현재 원소가 당첨자
  - Example
    - 3개의 작업
    - 당첨 번호: 300 ➔ C가 schedule 된다.




```
1   // counter: used to track if we've found the winner yet
2   int counter = 0;
3
4   // winner: use some call to a random number generator to
5   //         get a value, between 0 and the total # of tickets
6   int winner = getrandom(0, totaltickets);
7
8   // current: use this to walk through the list of jobs
9   node_t *current = head;
10
11  // loop until the sum of ticket values is > the winner
12  while (current) {
13      counter = counter + current->tickets;
14      if (counter > winner)
15          break; // found the winner
16      current = current->next;
17  }
18  // 'current' is the winner: schedule it...
```

Figure 9.1: **Lottery Scheduling Decision Code**

Proportional Share(4/5)

- 불공정성 분석
  - 가정: 2개의 작업, 같은 ticket, 같은 수행시간
  - U = C1/C2
    - C1: 첫번째 작업이 종료된 시간
    - C2: 두번째 작업이 종료된 시간
    - Example
      - C1=10, C2=20 ➔ U = 0.5 (불공정)
      - C1=20, C2=20 ➔ U = 1 (공정)
  - 작업이 충분한 시간 동안 실행 되어야 lottery scheduler는 원하는 결과에 가까워 짐

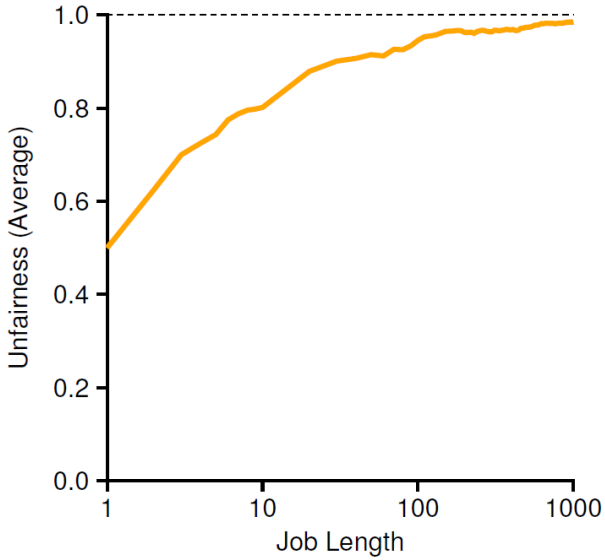


Figure 9.2: **Lottery Fairness Study**

Proportional Share(5/5)

- Lottery scheduling
  - 비 결정론적(random)
- Stride scheduling
  - 결정론적 fair-share scheduler
  - Stride : ticket의 수에 반비례
    - 임의의 큰 값을 ticket의 개수로 나눈 값
    - 가장 작은 pass 값을 가진 프로세스를 선택
    - 프로세스를 실행시킬 때마다 pass 값을 보폭만큼 증가
  - Example
    - Ticket – A: 100, B: 50, C: 250
    - Stride – 100, 200, 40 (divide 10000)
- 비 결정론적 방법은 프로세스 상태(CPU 사용현황, pass 값)를 유지할 필요가 없다.
  - Stride scheduling은 새로운 프로세스를 추가하기가 어렵다.
    - Pass 값을 정하기가 어렵기 때문

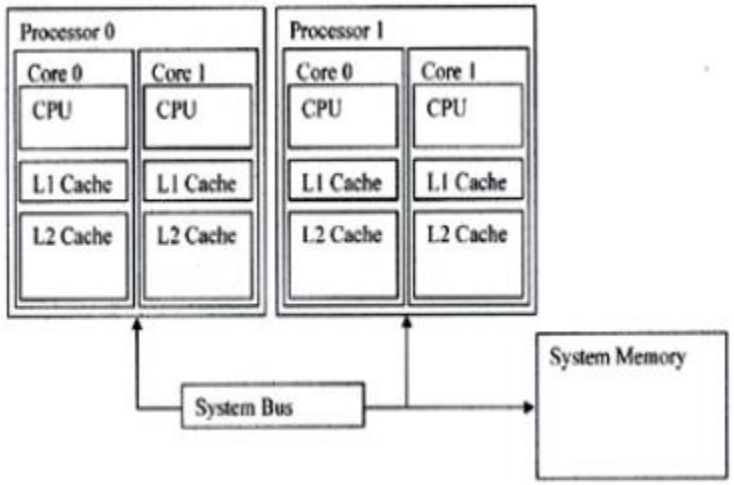
Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Figure 9.3: **Stride Scheduling: A Trace**

Multiprocessor Scheduling(1/5)

- 최근 Multicore, Multiprocessor 환경이 보편적
  - Multiprocessor
    - 하나의 시스템에 여러 개의 processor가 존재
  - Multicore
    - 하나의 processor chip에 여러 개의 core가 존재

- Multicore programming
  - 기존의 C 프로그램은 single core 환경에서 작성 되었기 때문에 core를 추가해도 더 빨리 실행되지 않는다.
    - Multi thread를 이용하여 여러 core에서 병렬적으로 실행되도록 수정해야 함
  - Scheduler가 multiple core, multiple processor를 handling 해야함



Multiprocessor Scheduling(2/5)

- 자주 사용하는 데이터는 속도가 빠른 cache에 저장
  - Temporal locality
    - 데이터가 한번 접근되면 가까운 미래에 다시 접근 될 가능성이 높다.
  - Spatial locality
    - 접근된 데이터의 주변의 데이터가 접근 될 가능성이 높다.
  - Problem: Cache coherence
    - CPU1에서 주소 A(값 D)를 memory에서 read ➔ 주소 변경 ➔ 값 D를 D'으로 변경 ➔ delayed write to memory ➔ CPU2 scheduling ➔ 주소 A 다시 read ➔ D'이 아닌 예전의 값 D를 read하여 cache coherence문제 발생
  - Solution: Bus snooping
    - Monitoring cache, cache의 내용이 변경되면 invalidate or update

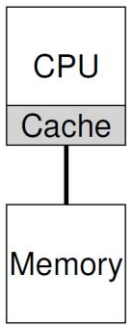


Figure 10.1: **Single CPU With Cache**

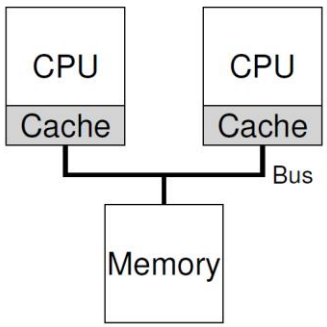


Figure 10.2: **Two CPUs With Caches Sharing Memory**

Multiprocessor Scheduling(3/5)

- 동기화
  - 공유되는 데이터의 올바른 연산 결과를 보장(갱신시)하기 위해 mutual exclusion을 보장하는 동기화 기법 사용
  - Example
    - 연결리스트에서 원소 하나를 삭제
    - Multithread 환경
      - 두개의 쓰레드가 동시에 List_Pop 함수에 진입
        - 헤드 원소를 두 번 삭제하고 같은 데이터 값을 두 번 반환

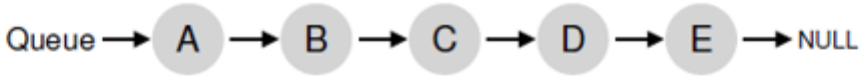
```
1   typedef struct __Node_t {
2       int              value;
3       struct __Node_t *next;
4   } Node_t;
5
6   int List_Pop() {
7       Node_t *tmp = head;          // remember old head ...
8       int value   = head->value;   // ... and its value
9       head        = head->next;    // advance head to next pointer
10      free(tmp);                   // free old head
11      return value;                // return value at head
12  }
```

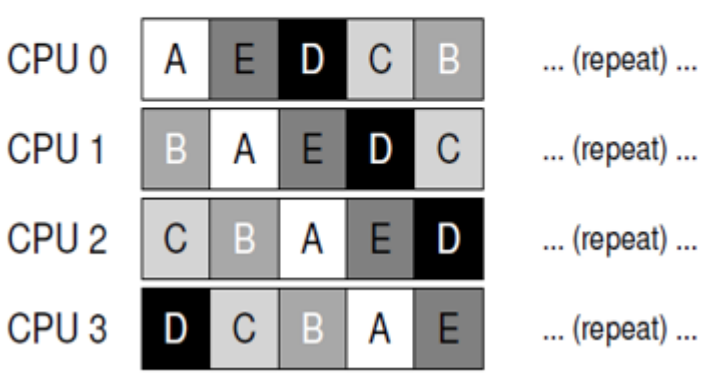
Figure 10.3: **Simple List Delete Code**

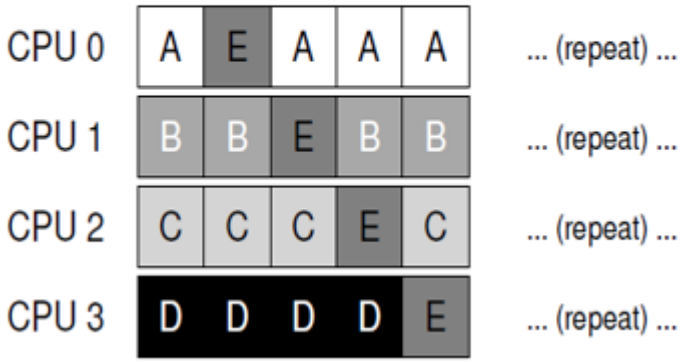
- Cache affinity
  - 프로세스가 수행될 때, cache와 TLB에 상당한 양의 정보를 올려 놓으므로, 다음 번에 수행될 때 동일한 CPU에서 실행되는 것이 유리

Multiprocessor Scheduling(4/5)

- SQMS (Single Queue Multiprocessor Scheduling) –BFS
  - 단일 프로세서 스케줄링의 기본 프레임워크를 그대로 사용
  - 장점
    - 단순함 (CPU가 2개라면 실행할 작업을 2개 선택)
  - 단점
    - Scalability : shared queue에 접근할 때 lock을 잡아 성능 저하
      - 경쟁이 증가(CPU 증가)할 수록 lock에 더 많은 시간 소모
    - Cache affinity
      - 복잡한 mechanism 필요

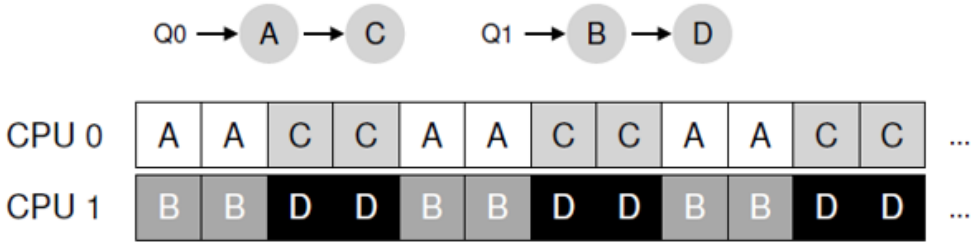




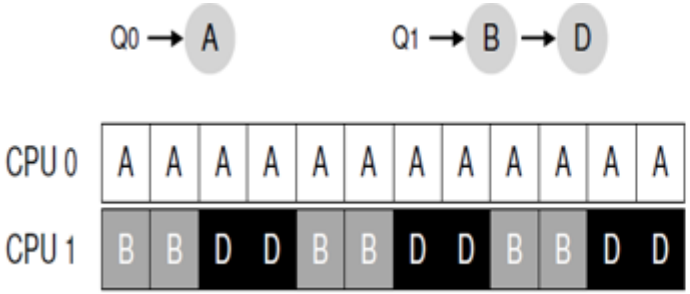


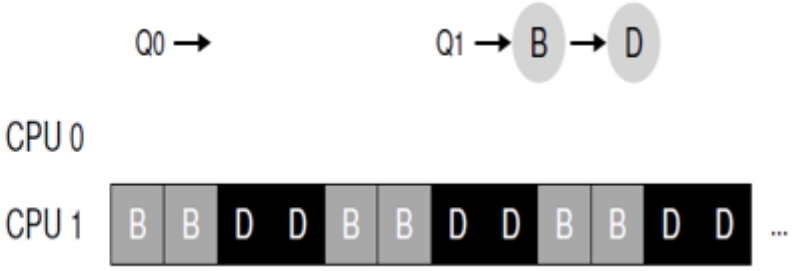
Multiprocessor Scheduling(5/5)

- MQMS (Multi-Queue Multiprocessor Scheduling) – O(1), CFS
  - CPU마다 queue를 하나씩 둔다.
  - 장점
    - Cache affinity, less lock contention



  - 단점
    - Load balancing을 위해 migration 필요
      - Work stealing: 작업의 개수가 낮은 queue가 다른 queue의 작업을 검사 (검사 주기 - tradeoff)





Q&A