Virtualizing the CPU

Limited Direct Execution & Scheduling

Liu yufeng

Fx_yfliu@163.com

Hunan University

- By time sharing the CPU, virtualization is achieved. However, there are a few challenges in building such virtualization machinery.

- The first is *performance*: how can we implement virtualization without adding excessive overhead to the system?

- The second is *control*: how can we run processes efficiently while retaining control over the CPU?

分时共享->虚拟化

这种虚拟化机制带来的挑战： 系统性能&OS对系统的掌控

关键问题：如何高效、可控地虚拟化 CPU

操作系统必须以高性能的方式虚拟化 CPU，同时保持对系统的控制。为此，需要硬件和操作系统支持。操作系统通常会明智地利用硬件支持，以便高效地实现其工作。

Basic Technique: Limited Direct Execution

- The “**direct execution**” part of the idea is simple: just run the program directly on the CPU.

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	

？哪个系统调用

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

？这种情况会有什么问题

fork（），

OS一旦把控制权交给用户进程

可能就失去了控制（CPU和其他硬件）

- This approach gives rise to a few problems:
- The first: how can the OS make sure the program doesn’t do anything that we don’t want it to do, while still running it efficiently?
- The second: how does the operating system stop a process from running and switch to another one, thus implementing the time sharing?

Problem #1: Restricted Operations

- Direct execution has the obvious advantage of being fast.

- But it introduces a problem: **what if the process wishes to perform some kind of restricted operation**, such as:

  - issuing an I/O request to a disk,

  - gaining access to more system resources such as CPU or memory?

关键问题：如何执行受限制的操作

一个进程必须能够执行 I/O 和其他一些受限制的操作，但又不能让进程完全控制系统。操作系统和硬件如何协作实现这一点？

- The approach is to introduce a new processor mode, known as **user mode**.

- In contrast to user mode is **kernel mode**, which the operating system (or kernel) runs in.

- What should a user process do when it wishes to perform some kind of privileged operation?

  - All modern hardware provides the ability for user programs to perform a **system call**.

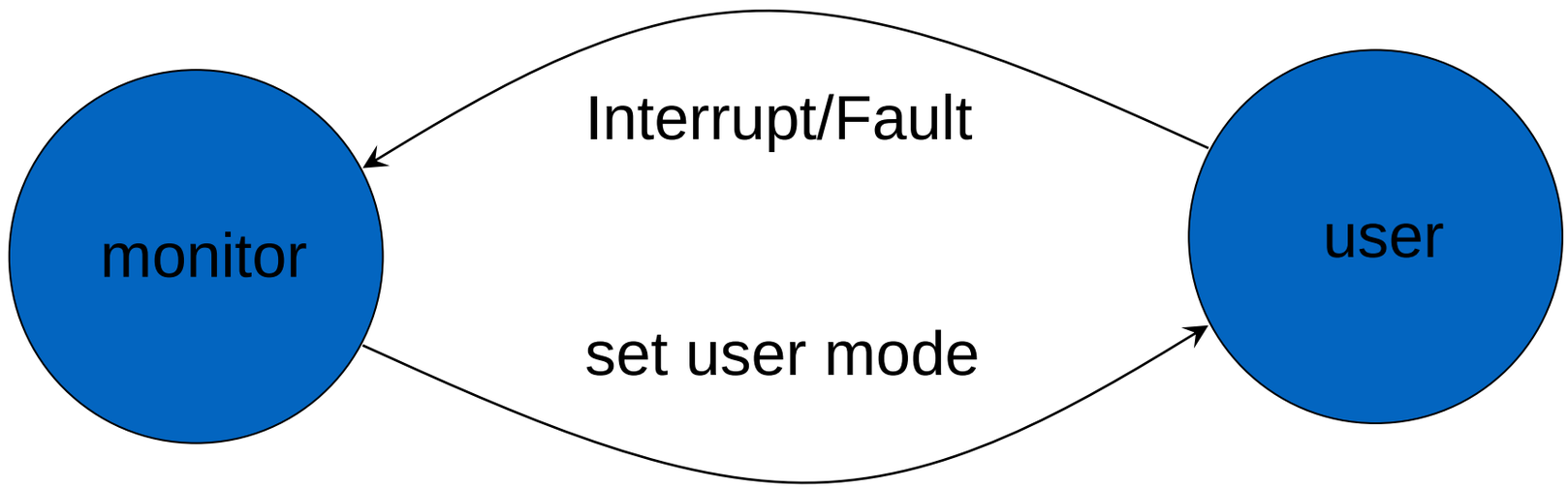
两种模态：用户态和内核态

用户程序只能通过OS的系统调用进入到内核态，使用特权操作

双重模式操作

- OS必须提供硬件支持用来区分至少以下两种操作模式
  - 用户模式（user mode） — 代表用户在执行
  - 内核模式（kernel mode）— 代表OS在执行
- 内核空间存放的是操作系统内核代码和数据，是被所有程序共享的，在程序中修改内核空间中的数据不仅会影响操作系统本身的稳定性，还会影响其他程序，这是非常危险的行为，所以操作系统禁止用户程序直接访问内核空间。
- 用户要想访问内核空间，必须借助操作系统提供的 API 函数，执行内核提供的代码，让内核自己来访问，这样才能保证内核空间的数据不会被随意修改，才能保证操作系统本身和其他程序的稳定性。
- 用户程序调用系统 API 函数称为系统调用（System Call）；发生系统调用时会暂停用户程序，转而执行内核代码（内核也是程序），访问内核空间，这称为内核模式（Kernel Mode）。

- 当中断或错误发生时，硬件自动切换成监督程序模式（monitor mode）



- 所有的I/O指令都是特权指令
- 必须确保用户程序永远无法以user模式获得计算机的控制权

General-System Architecture

- I/O指令是特殊指令，用户程序该如何执行I/O操作？
- 系统调用 — 进程用来向OS请求服务的方式
  - 通常采用陷阱的方式来进入中断向量
  - 控制通过中断向量传递到OS内的服务例程，这时模式位（mode bit）设为监督模式
  - 操作系统检验参数的合法性，执行请求，并将控制返回给系统调用后面的指令

用户态切换到内核态的3种方式

1）系统调用 这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。

2）异常 当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

3）外围设备的中断 当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

System Call

- To execute a system call, a program must execute a special **trap** instruction, which simultaneously jumps into the kernel and raises the privilege level to kernel mode.

- When finished, the OS calls a special **return-from-trap** instruction, which returns into the calling user program while simultaneously reducing the privilege level back to user mode.

- The hardware needs to save the caller’s registers in order to be able to return correctly when the OS issues the return-from-trap instruction.（执行陷阱时，硬件需要小心，因为它必须确保存储足够的调用者寄存器，以便在操作 系统发出从陷阱返回指令时能够正确返回。）

- On x86, the processor will push the *program counter*, *flags*, and a few other *registers* onto a per-process **kernel stack**.（例如，在 x86 上，处理器会将程序计数器、标志和其他一些寄存器推送到每个进程的内核栈（kernel stack）上。从返回陷阱将从栈弹出这 些值，并恢复执行用户模式程序）

How does the trap know which code to run inside the OS?

- The kernel does so by setting up a **trap table** at boot time in privileged (kernel) mode.

- One of the first things the OS does is to tell the hardware what code to run when certain exceptional events (a hard-disk interrupt, a keyboard interrupt, or a system call) occur.

- The OS informs the hardware of the locations of these **trap handlers**.

- To specify the exact system call, a **system-call number** is usually assigned to each system call.

- The instruction to set up the trap tables is a **privileged** operation.

？为什么设置陷进表是特权操作

这是OS要实现对系统的控制
他自己的东西，当然不允许用户修改

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of... syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call
		trap into OS
	save regs (to kernel stack)	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to PC after trap	
		return from main
		trap (via `exit()`)
Free memory of process		
Remove from process list		

Figure 6.2: **Limited Direct Execution Protocol**

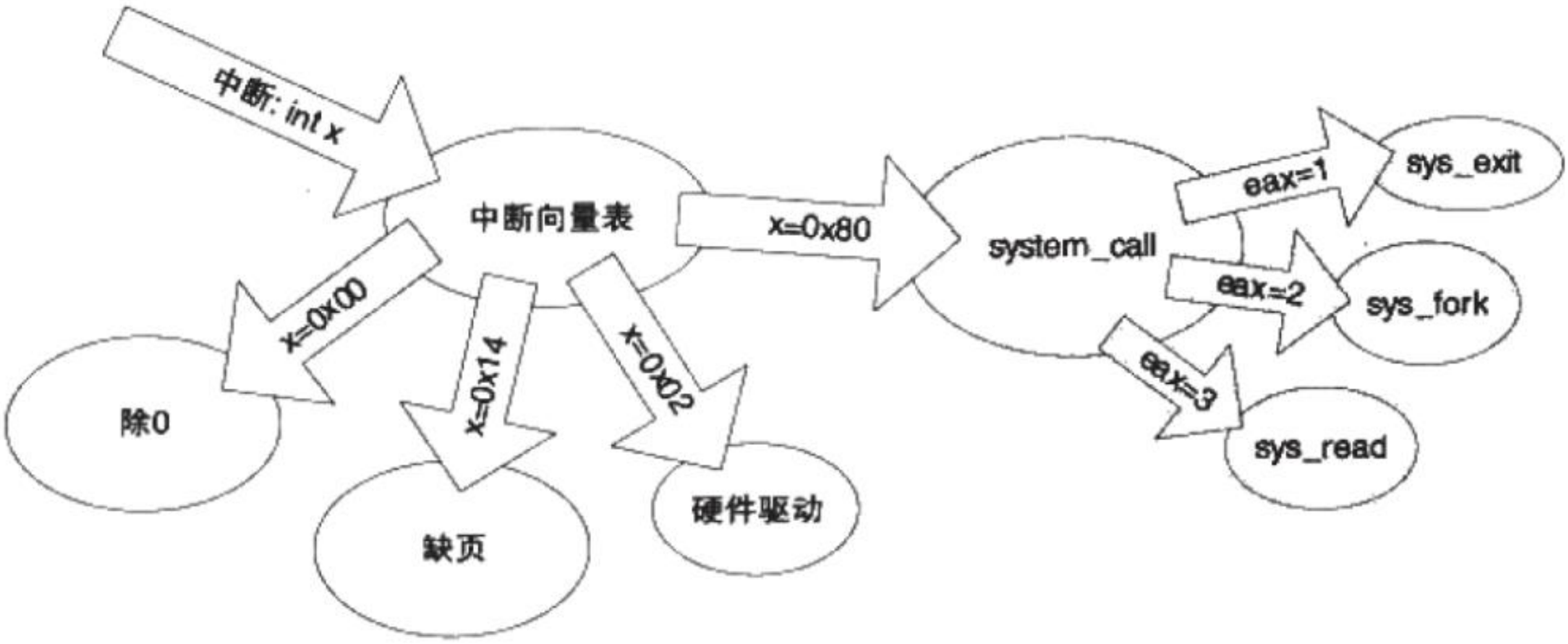
？这种情况解决了什么问题，还有什么问题

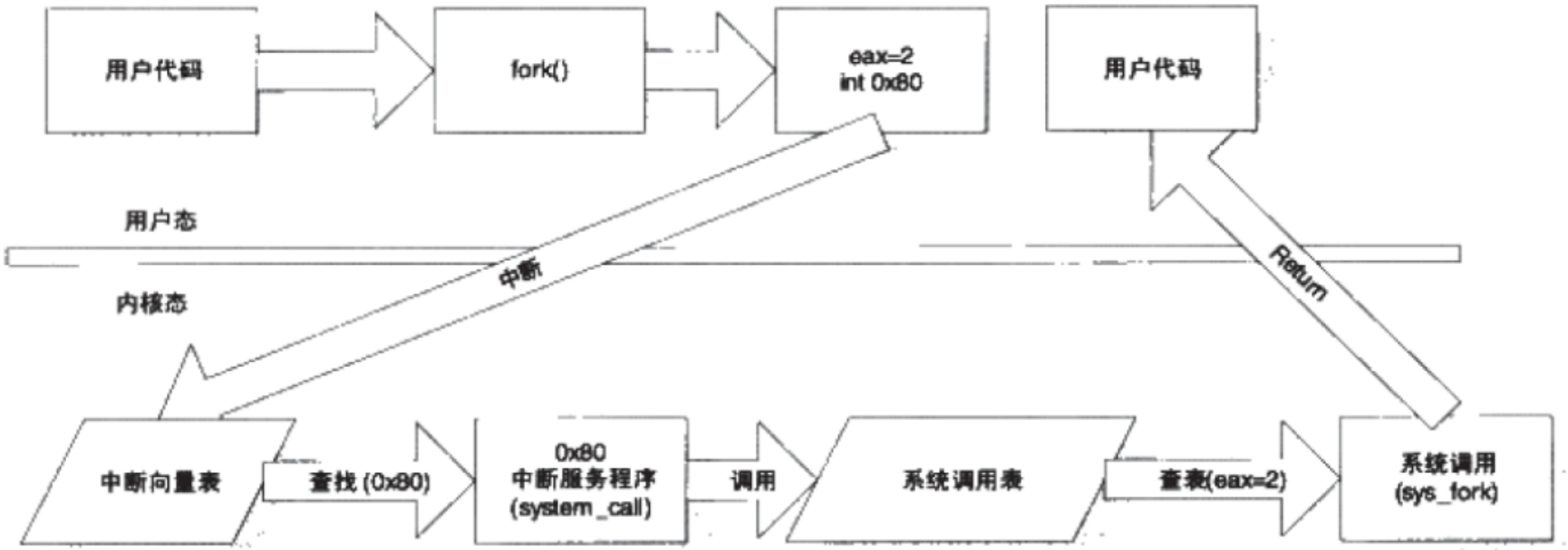
OS实现了特权控制
但是CPU呢？万一main无休止呢？

补充：为什么系统调用看起来像过程调用

你可能想知道，为什么对系统调用的调用（如 open()或 read()）看起来完全就像 C 中的典型过程调用。也就是说，如果它看起来像一个过程调用，系统如何知道这是一个系统调用，并做所有正确的事情？原因很简单：它是一个过程调用，但隐藏在过程调用内部的是著名的陷阱指令。更具体地说，当你调用 open()（举个例子）时，你正在执行对 C 库的过程调用。其中，无论是对于 open()还是提供的其他系统调用，库都使用与内核一致的调用约定来将参数放在众所周知的位置（例如，在栈中或特定的寄存器中），将系统调用号也放入一个众所周知的位置（同样，放在栈或寄存器中），然后执行上述的陷阱指令。库中陷阱之后的代码准备好返回值，并将控制权返回给发出系统调用的程序。因此，C 库中进行系统调用的部分是用汇编手工编码的，因为它们需要仔细遵循约定，以便正确处理参数和返回值，以及执行硬件特定的陷阱指令。现在你知道为什么你自己不必写汇编代码来陷入操作系统了，因为有人已经为你写了这些汇编。


用户程序
int open(const char *pathname, int flags, mode_t mode)
系统调用名称
参数1
参数2
参数3
参数4
参数5
参数6
glibc
DO_CALL (syscall_name, args)
保存寄存器
%ebx
%ecx
%edx
%esi
%edi
%ebp
系统调用名称转为系统调用号
%eax
ENTER_KERNEL int $0x80
内核
IA32_SYSCALL_VECTOR
entry_INT80_32
保存寄存器到内核数据结构
struct pt_regs
orig_ax
bx
cx
dx
si
di
bp
在系统调用表里面找到相应的函数
sys_call_table[]
0
1
2
系统调用实现函数
参数1
参数2
参数3
参数4
参数5
参数6
struct pt_regs
返回值
ax


中断: int x
中断向量表
x=0x80
system_call
eax=1
sys_exit
eax=2
sys_fork
eax=3
sys_read
x=0x00
除0
x=0x14
缺页
x=0x02
硬件驱动


用户代码
fork()
eax=2
int 0x80
用户代码
用户态
内核态
中断
Return
中断向量表
查找 (0x80)
0x80
中断服务程序
(system_call)
调用
系统调用表
查表(eax=2)
系统调用
(sys_fork)

Problem #2: Switching Between Processes

- If a process is running on the CPU, this means the OS is not running. **If the OS is not running, how can it do anything at all?** (hint: it can’t)

- There is clearly no way for the OS to take an action if it is not running on the CPU.

关键问题：如何重获 CPU 的控制权

操作系统如何重新获得 CPU 的控制权（regain control），以便它可以在进程之间切换？

A Cooperative Approach: Wait For System Calls

- The OS **trusts** the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU.

- Systems often include an explicit `yield` system call, which does nothing except to transfer control to the OS.

- Applications also transfer control to the OS when they do something illegal (by generating a trap to the OS).

关键问题：如何在没有协作的情况下获得控制权

即使进程不协作，操作系统如何获得 CPU 的控制权？操作系统可以做什么来确保流氓进程不会占用机器？

A Non-Cooperative Approach: The OS Takes Control

- The answer is **timer interrupt**. A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the currently running process is halted, and a pre-configured **interrupt handler** in the OS runs.

Saving and Restoring Context

- When the OS has regained control, it needs to decide whether to continue running the currently-running process, or switch to a different one. This is made by the **scheduler**.（调度器决定哪个进程将使用CPU）

- If the decision is made to switch, the OS then performs a **context switch**.（完成不同进程的切换）

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) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

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

？ 中断和陷进对寄存器内容如何保存

OS完美实现了控制！

```
# Save old registers
movl 4(%esp), %eax  # put old ptr into eax
popl 0(%eax)        # save the old IP
movl %esp, 4(%eax)  # and stack
movl %ebx, 8(%eax)  # and other registers
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)

# Load new registers
movl 4(%esp), %eax  # put new ptr into eax
movl 28(%eax), %ebp # restore other registers
movl 24(%eax), %edi
movl 20(%eax), %esi
movl 16(%eax), %edx
movl 12(%eax), %ecx
movl 8(%eax), %ebx
movl 4(%eax), %esp  # stack is switched here
pushl 0(%eax)       # return addr put in place
ret                 # finally return into new ctxt
```

图 6.1　xv6 的上下文切换代码

补充：上下文切换要多长时间

你可能有一个很自然的问题：上下文切换需要多长时间？甚至系统调用要多长时间？如果感到好奇，有一种称为 lmbench [MS96]的工具，可以准确衡量这些事情，并提供其他一些可能相关的性能指标。

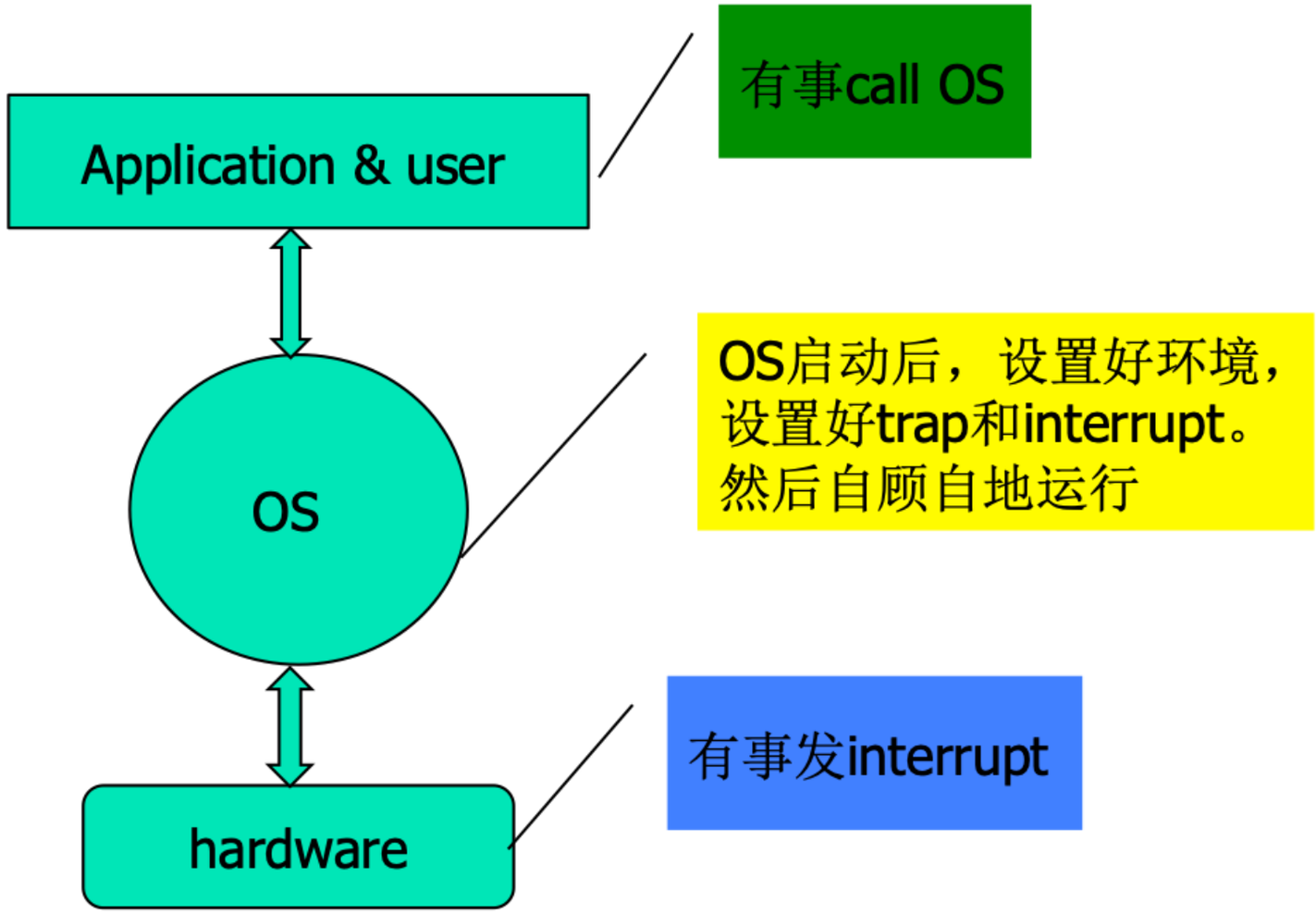
随着时间的推移，结果有了很大的提高，大致跟上了处理器的性能提高。例如，1996 年在 200-MHz P6 CPU 上运行 Linux 1.3.37，系统调用花费了大约 4μs，上下文切换时间大约为 6μs[MS96]。现代系统的性能几乎可以提高一个数量级，在具有 2 GHz 或 3 GHz 处理器的系统上的性能可以达到亚微秒级。

应该注意的是，并非所有的操作系统操作都会跟踪 CPU 的性能。正如 Ousterhout 所说的，许多操作系统操作都是内存密集型的，而随着时间的推移，内存带宽并没有像处理器速度那样显著提高[O90]。因此，根据你的工作负载，购买最新、性能好的处理器可能不会像你希望的那样加速操作系统。

1毫秒（ms）等于一千分之一秒
1微秒（us）等于一千分之一毫秒
1纳秒（ns）等于一千分之一微秒

执行典型指令	1/1,000,000,000 秒 =1 纳秒
从一级缓存中读取数据	0.5 纳秒
分支预测错误	5 纳秒
从二级缓存中读取数据	7 纳秒
互斥锁定 / 解锁	25 纳秒
从主存储器中读取数据	100 纳秒
在 1Gbps 的网络中发送 2KB 数据	20,000 纳秒
从内存中读取 1MB 数据	250,000 纳秒
从新的磁盘位置读取数据 (寻轨)	8,000,000 纳秒
从磁盘中读取 1MB 数据	20,000,000 纳秒
在美国向欧洲发包并返回	150 毫秒 =150,000,000 纳秒

OS中断驱动运行模式


Application & user
OS
hardware
有事call OS
OS启动后，设置好环境，
设置好trap和interrupt。
然后自顾自地运行
有事发interrupt

Summary

- The CPU should support at least two modes of execution: a restricted **user mode** and a privileged (non-restricted) **kernel mode**.
- Typical user applications run in user mode, and use a **system call** to **trap** into the kernel to request operating system services.
- The trap instruction saves register state carefully, changes the hardware status to kernel mode, and jumps into the OS to a pre-specified destination: the **trap table**.
- When the OS finishes servicing a system call, it returns to the user program via another special **return-from-trap** instruction, which reduces privilege and returns control to the instruction after the trap that jumped into the OS.
- The trap tables must be set up by the OS at boot time, and make sure that they cannot be readily modified by user programs. All of this is part of the **limited direct execution** protocol which runs programs efficiently but without loss of OS control.
- Once a program is running, the OS must use hardware mechanisms to ensure the user program does not run forever, namely the **timer interrupt**. This approach is a **non-cooperative** approach to CPU scheduling.
- Sometimes the OS, during a timer interrupt or system call, might wish to switch from running the current process to a different one, a low-level technique known as a **context switch**.

硬件至少提供两种模态
用户态和内核态

用户应用程序通过OS
提供的系统调用trap
到内核态得以执行

OS会提前构造一张trap表
记录每一个系统调用入口

return-from-trap指令
实现从内核态到用户态

trap表OS引导过程中设置
且只能OS访问

OS利用定时器
确保获得控制权，
而执行CPU调度

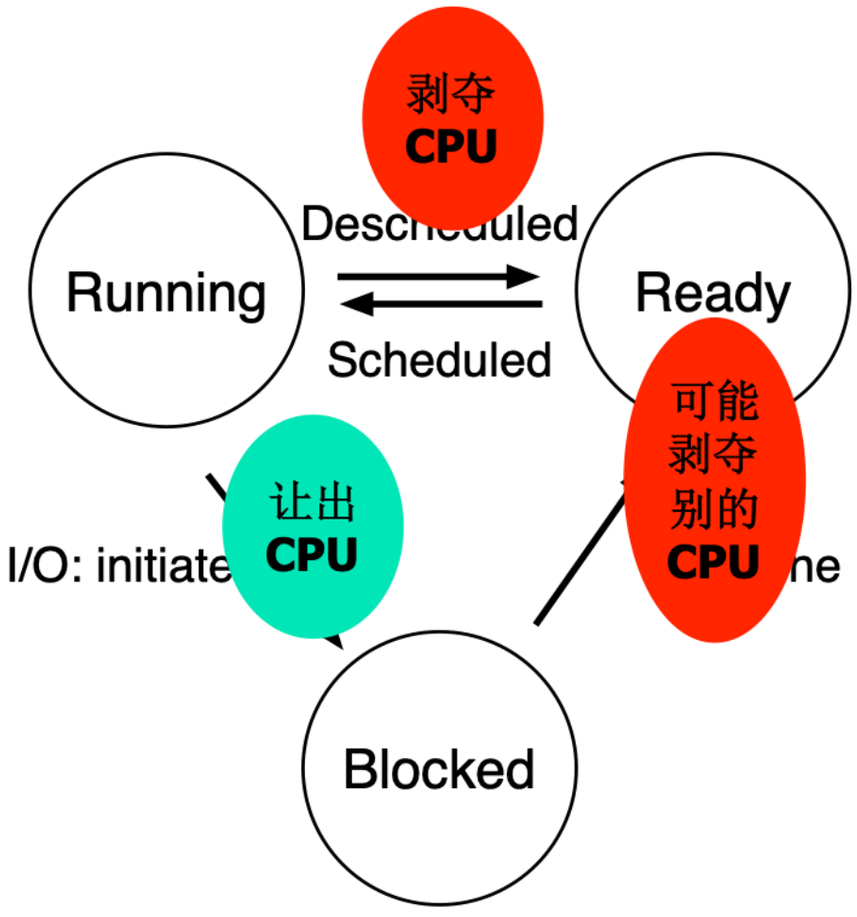
定时器中断、系统调用
OS都有可能要做进程
上下文切换

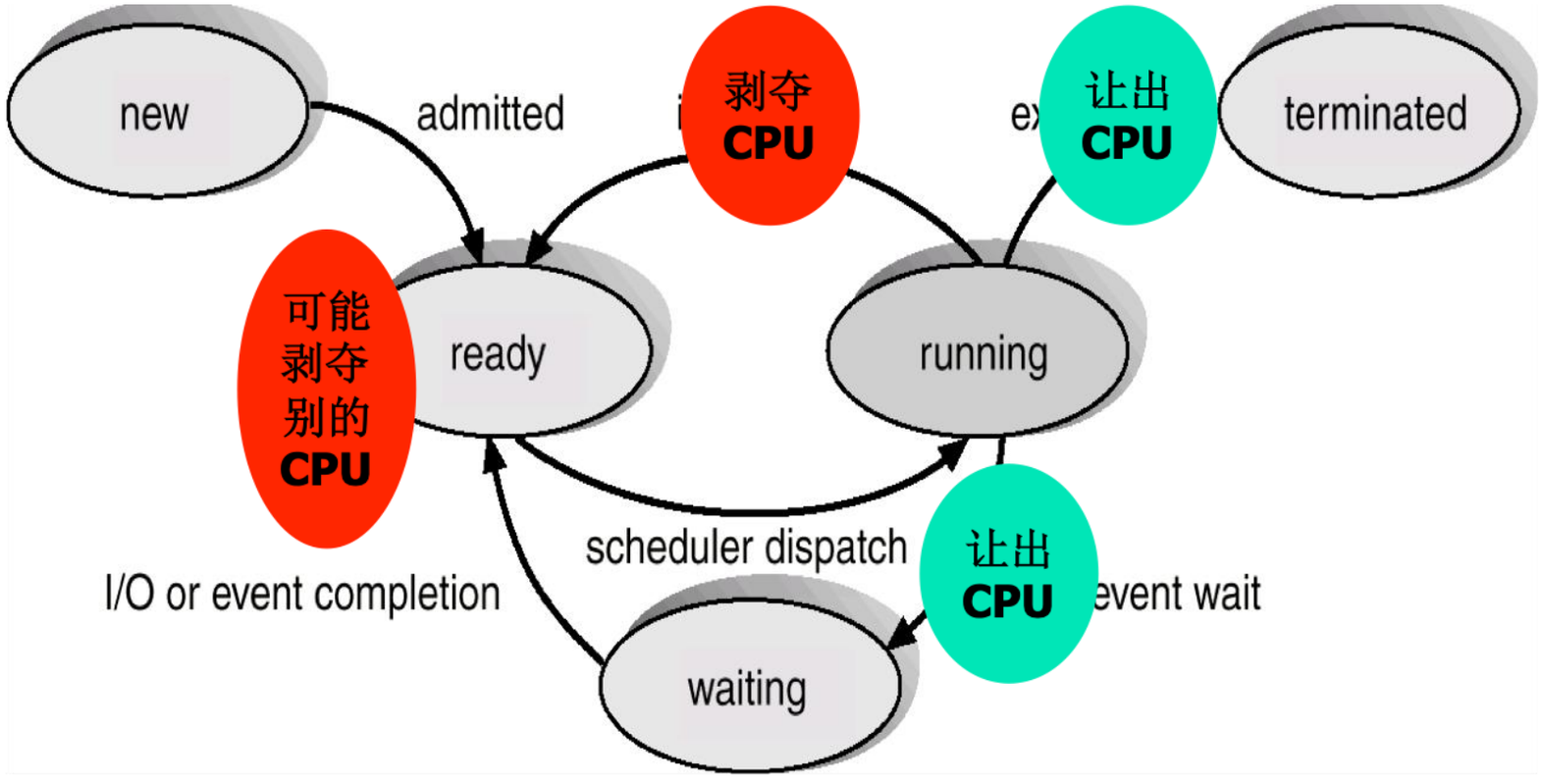
Scheduling

关于调度

- **Why?**
  - 虚拟化，运行的进程数>>CPU个数
- **Possible?**
  - CPU的速度>>用户的速度
  - CPU与I/O可以并发
- **Principle**
  - 让CPU不停地受控忙，公平地做有用功，考虑任务的轻重缓急

调度的时机





？什么时候调度　　什么时候CPU可能会空，什么时候有更紧急任务

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  3. Switches from waiting to ready.
  4. Terminates.
- Scheduling under 1 and 4 is *nonpreemptive*.
- All other scheduling is *preemptive*.

关键问题：如何开发调度策略

我们该如何开发一个考虑调度策略的基本框架？什么是关键假设？哪些指标非常重要？哪些基本方法已经在早期的系统中使用？

Workload Assumptions

- We first make a number of assumptions about the processes (called the **workload**).

- The workload assumptions are mostly unrealistic, but we will relax them.

Workload Assumptions

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

Scheduling Metrics

- **Scheduling metric**. A metric is just something that we use to measure something.

- **Turnaround time** of a job is the time at which the job completes minus the time at which the job arrived in the system.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

- Turnaround time is a **performance** metric, as compared to the metric **fairness**. **Performance and fairness are often at odds in scheduling**.

First In, First Out (FIFO)

First Come, First Served (FCFS)

- Imagine three jobs arrive in the system, A, B, and C, at roughly the same time ($T_{arrival} = 0$). Assume they all arrived simultaneously, and each job runs for 10 seconds. What is the **average turnaround time** for these jobs?

- **(10 + 20 + 30) / 3 = 20**

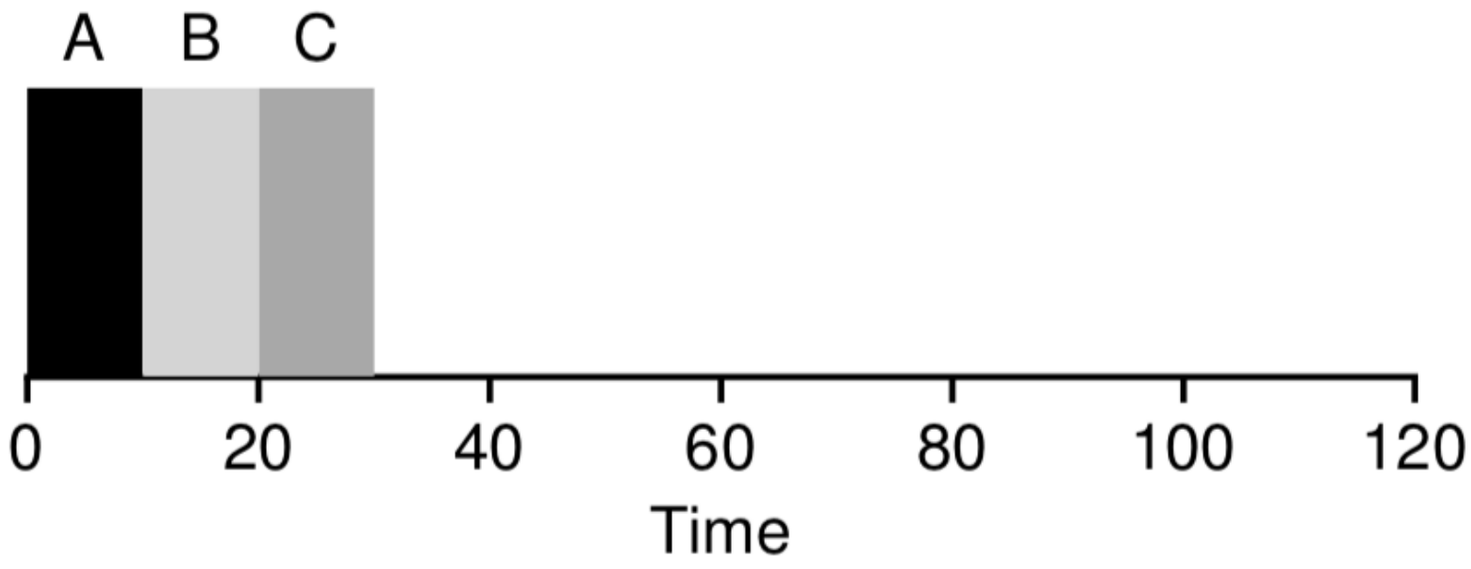


Figure 7.1: **FIFO Simple Example**

- Assume three jobs (A, B, and C), but this time A runs for 100 seconds while B and C run for 10 each.

- **(100 + 110 + 120) / 3 = 110**

- This problem is referred to as the **convoy effect**（**护航效应**）, where a number of relatively-short consumers of a resource get queued behind a heavyweight resource consumer.

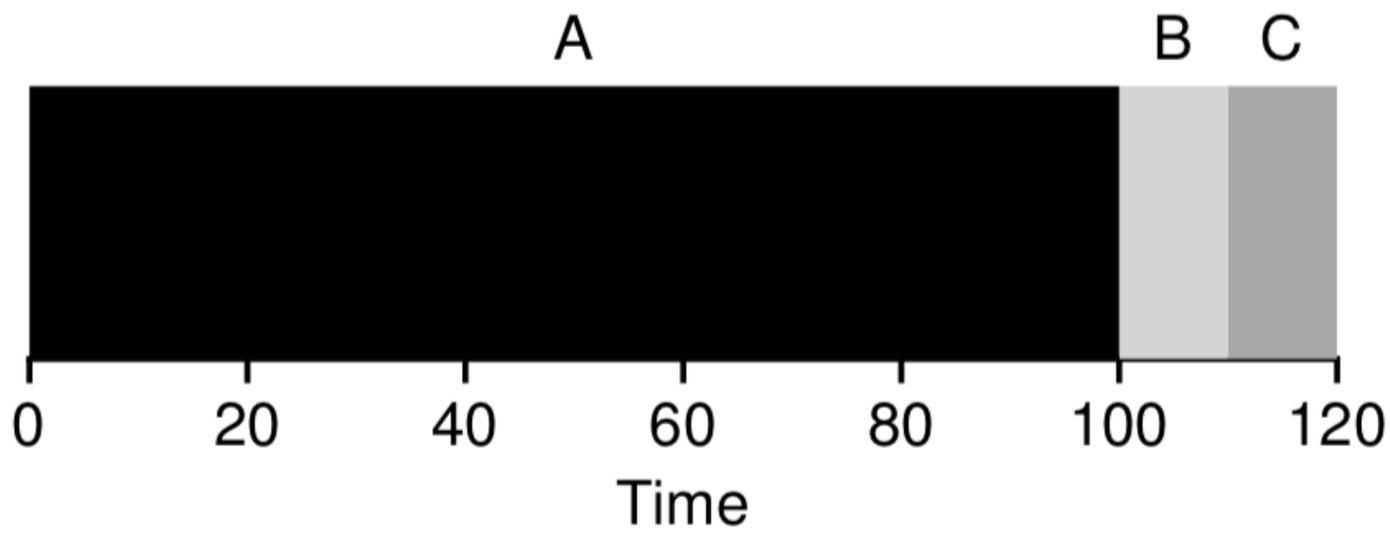


Figure 7.2: **Why FIFO Is Not That Great**

Shortest Job First (SJF)

- Shortest Job First (SJF) policy runs the shortest job first, then the next shortest, and so on.
- **(10 + 20 + 120) / 3 = 50**

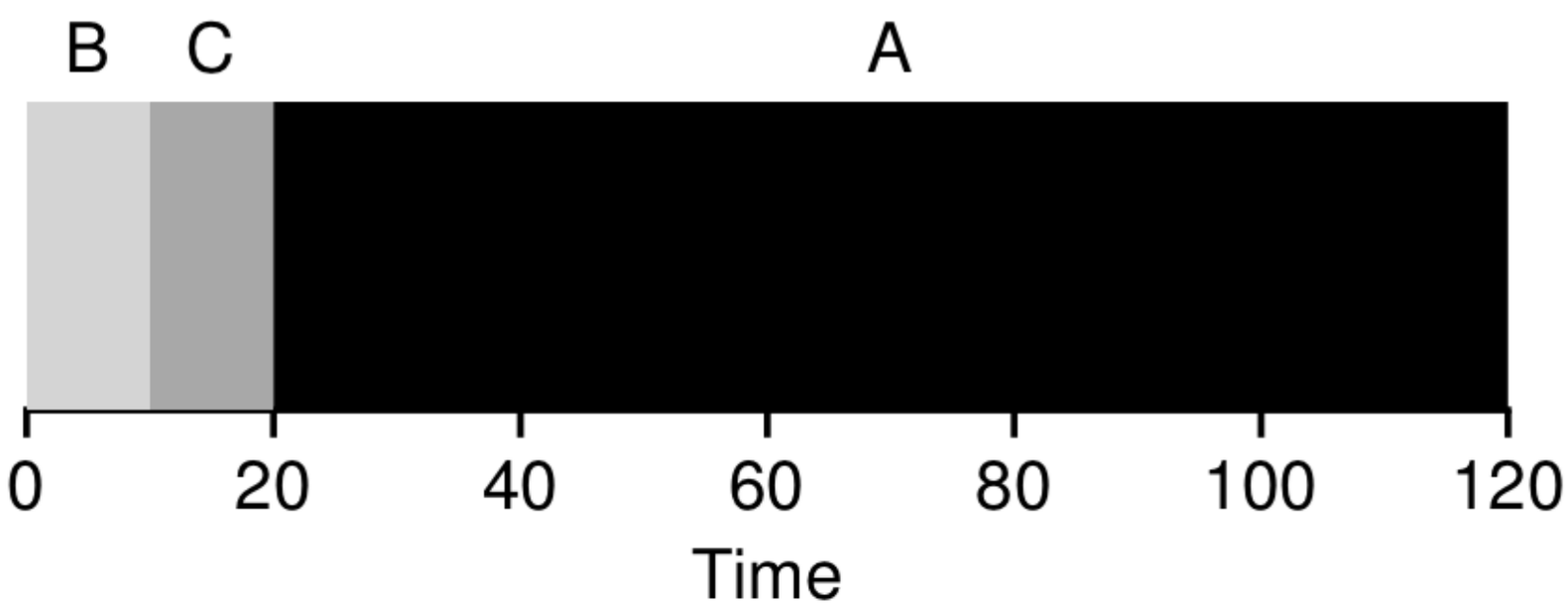


Figure 7.3: **SJF Simple Example**

- If jobs can **arrive at any time** instead of all at once. **What problems does this lead to**?

- Assume A arrives at t = 0 and needs to run for 100 seconds, whereas B and C arrive at t = 10 and each need to run for 10 seconds.

- **(100+(110−10)+(120−10)) / 3 = 103.33**

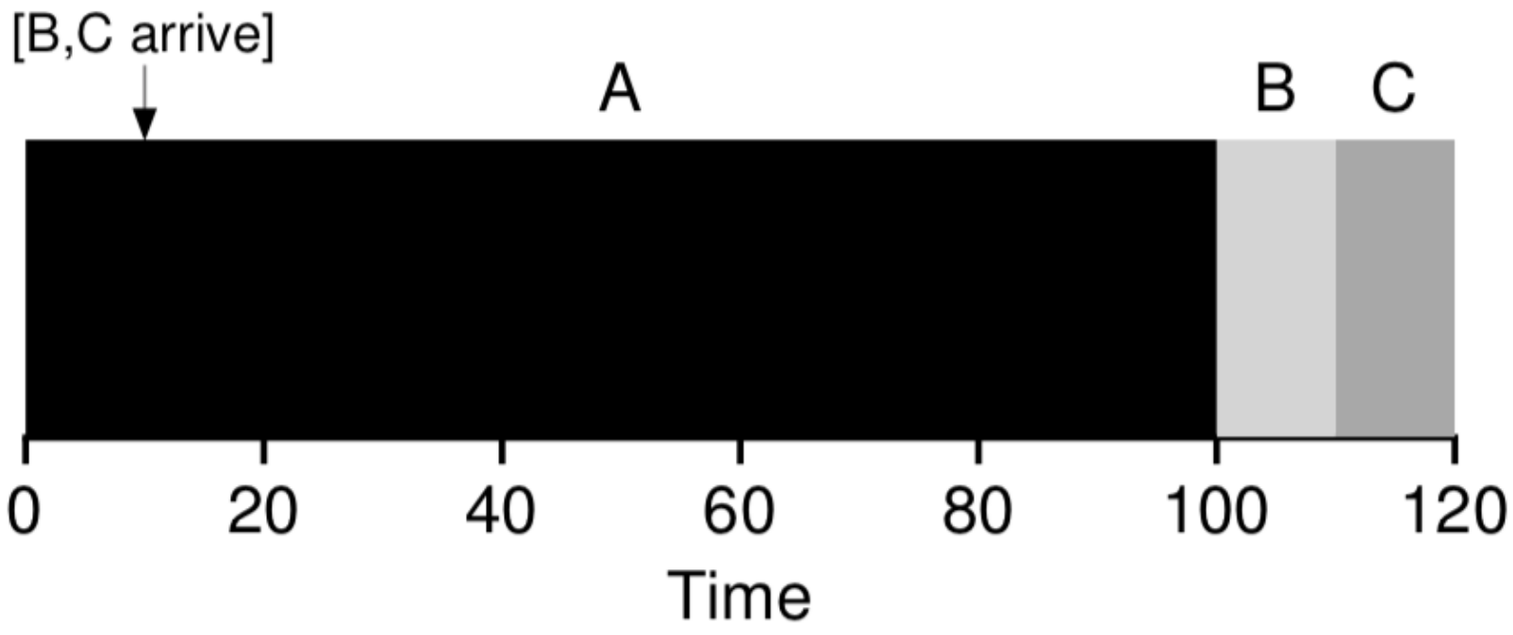


Figure 7.4: **SJF With Late Arrivals From B and C**

Shortest Time-to-Completion First (STCF)

Preemptive Shortest Job First，PSJF

- To address the problem of SJF, we need to relax assumption 3 (that jobs must run to completion).

- Given our previous discussion about *timer interrupts* and *context switching*, the scheduler it can **preempt** a job and decide to run another job.

- SJF by definition is a **non-preemptive** scheduler, and thus suffers from the problems described above.

- Add preemption to SJF, known as the **Shortest Time-to-Completion First** (**STCF**) or **Preemptive Shortest Job First** (**PSJF**) scheduler.

- **((120−0)+(20−10)+(30−10)) / 3 = 50**

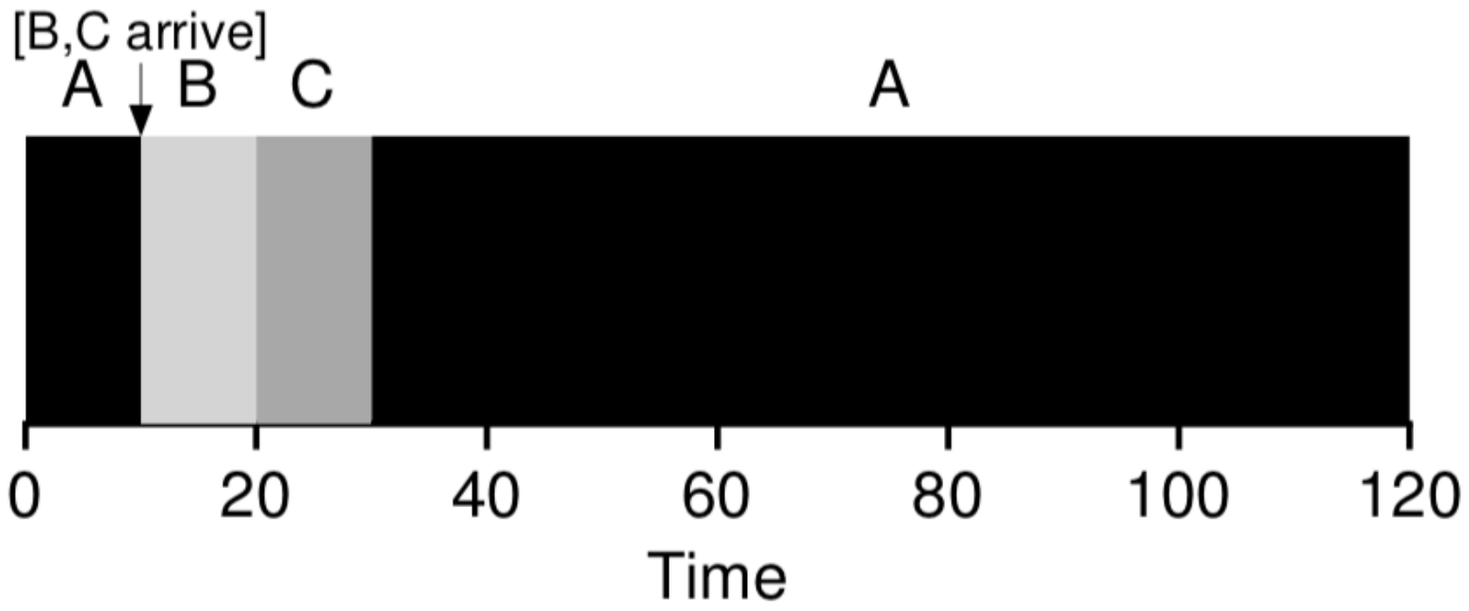


Figure 7.5: **STCF Simple Example**

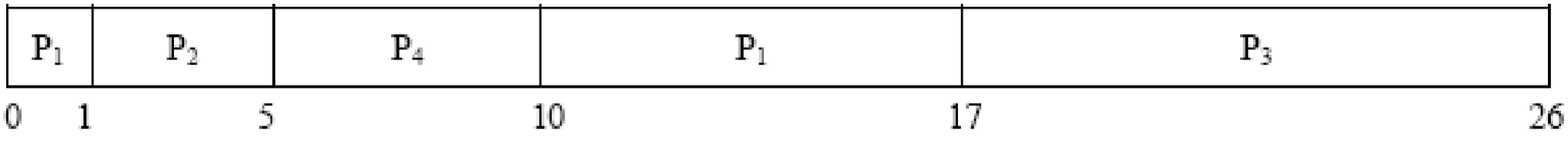
Example of Preemptive SJF

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- **SJF (preemptive)**

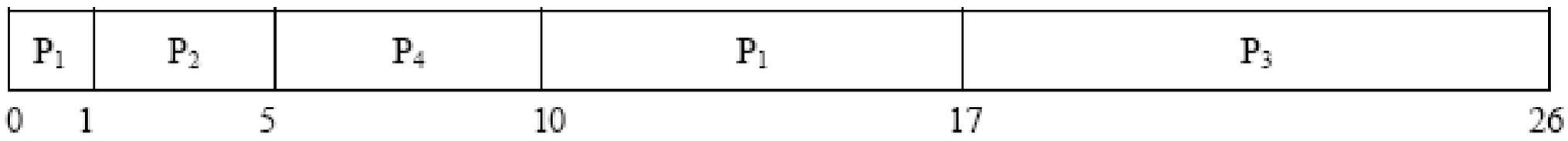


- Turnaround time = Exit time - Arrival time
- Waiting time = Turnaround time - Burst time（在就绪队列中的时间）

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- **SJF (preemptive)**



- Turnaround time = Exit time - Arrival time
- turnarouad time=((17-0)+(5-1)+(26-2)+(10-3))/4=13
- Waiting time = Turnaround time - Burst time
- Average waiting time = ((10-1)+(1-1)+(17-2)+(5-3))/4=6.5

A New Metric: Response Time

- If we knew job lengths, and our only metric was turnaround time, STCF would be a great policy.

- However, for time-shared machines, a new metric was needed: **response time**.

- We define response time as the time from when the job arrives in a system to the first time it is scheduled.

$$T_{response} = T_{firstrun} - T_{arrival}$$

- For example, with A arriving at time 0, and B and C at time 10, the response time of each job is: 0 for job A, 0 for B, and 10 for C (average: 3.33).

- STCF are bad for *response time* and *interactivity*, although great for *turnaround time*.

Round Robin

- To solve this problem, we introduce a new scheduling algorithm, classically referred to as **Round-Robin** (**RR**) scheduling.

- The basic idea is simple: instead of running jobs to completion, RR runs a job for a **time slice** and then switches to the next job in the run queue.

- It repeatedly does so until the jobs are finished. RR is sometimes called **time-slicing**.

- Assume three jobs A, B, and C arrive at the same time, and each wish to run for 5 seconds.

- The average response time of RR is: **(0 + 1 + 2) / 3= 1（时间片为1）**; for SJF, it is: **(0 + 5 + 10) / 3 = 5**.

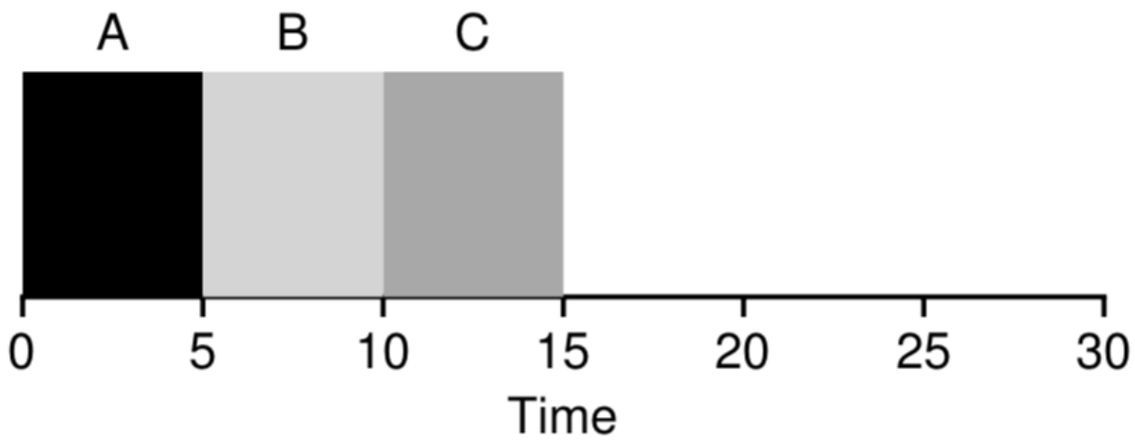


Figure 7.6: **SJF Again (Bad for Response Time)**

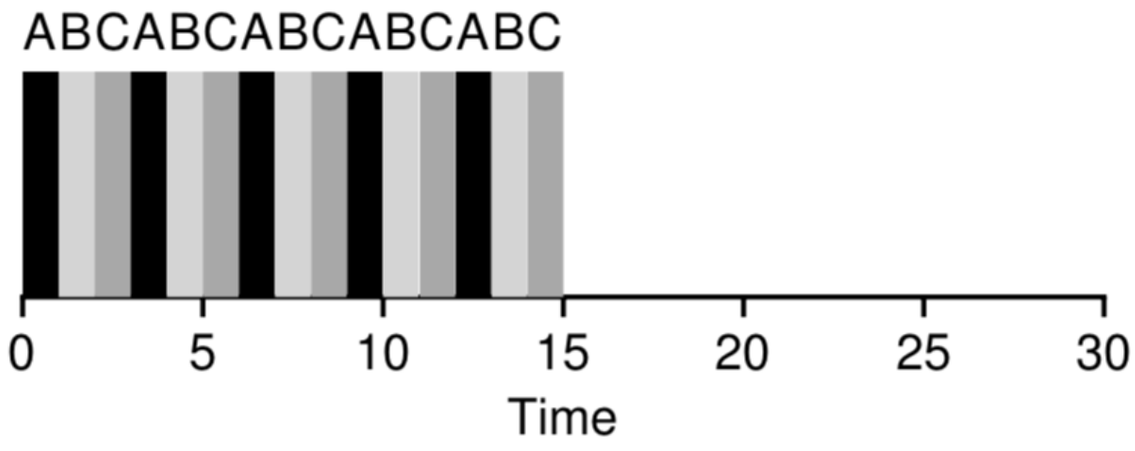


Figure 7.7: **Round Robin (Good For Response Time)**

What about the turnaround time for RR?

- Look at our example above again. A, B, and C, each running for 5 seconds, arrive at the same time, with a 1-second time slice.

- We can see from the picture above that A finishes at 13, B at 14, and C at 15, for an average of **14** (as compared to **10** of SJF).

You can’t have your cake and eat it too!

- RR is indeed one of the **worst** policies if *turnaround time* is our metric, even worse than simple FIFO in many cases.

- Any policy (such as RR) that is **fair**, will perform poorly on metrics such as turnaround time.

- This type of **trade-off** is common in systems.

Incorporating I/O

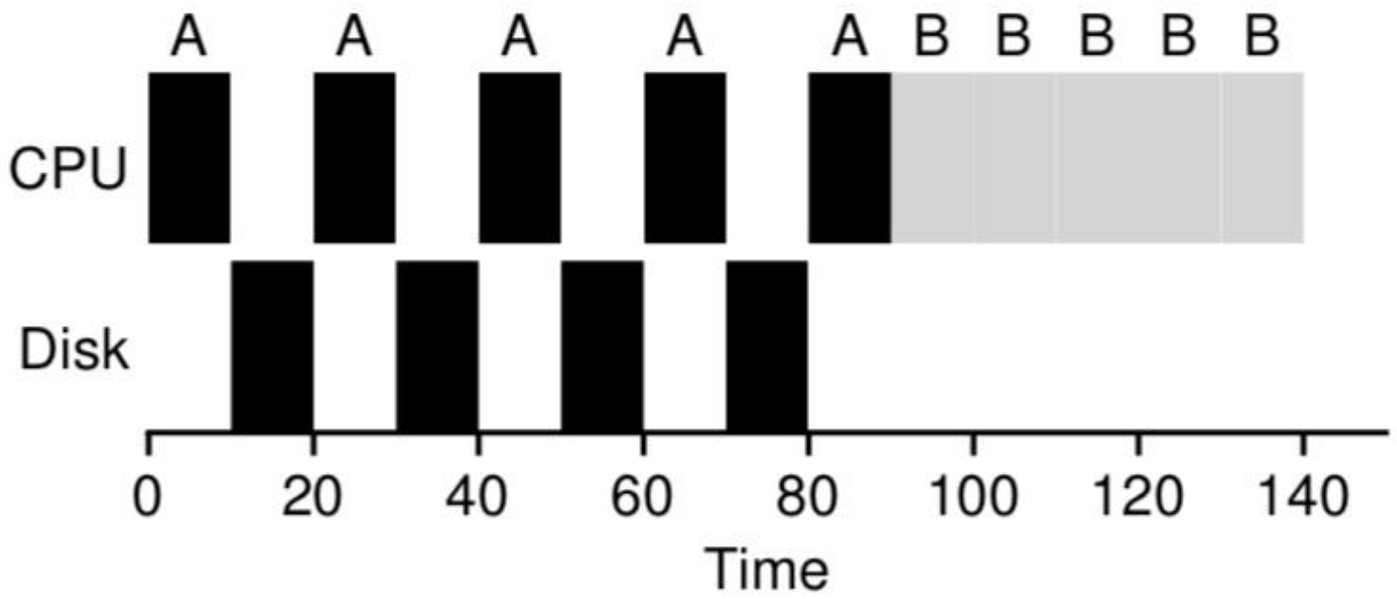


Figure 7.8: **Poor Use Of Resources**

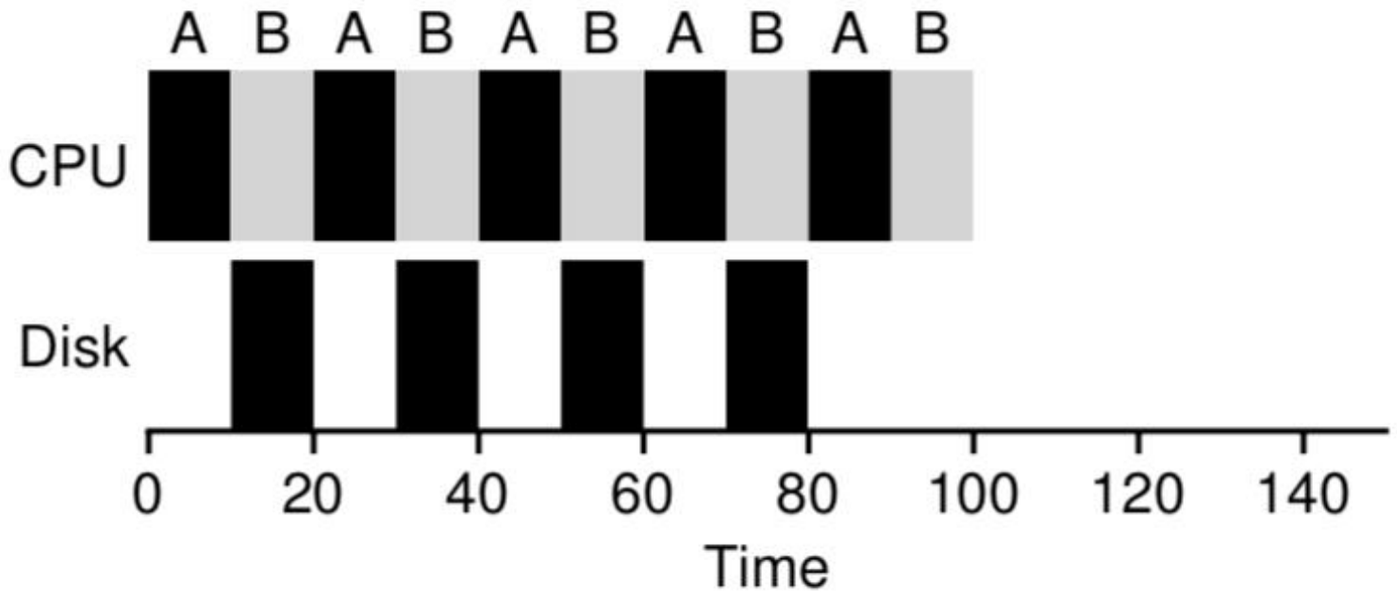


Figure 7.9: **Overlap Allows Better Use Of Resources**

Summary

- In fact, in a general-purpose OS, **the OS usually knows very little about the length of each job**.

- Thus, how can we build an approach that behaves like SJF/STCF without such *a priori* knowledge?

- Further, how can we incorporate some of the ideas we have seen with the RR scheduler so that response time is also quite good?

Determining Length of Next CPU Burst

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.

1. t_n = actual lenght of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n.$$

Examples of Exponential Averaging

- α =0
  - Tn+1 = Tn
  - Recent history does not count.
- α =1
  - Tn+1 = tn
  - Only the actual last CPU burst counts.
- If we expand the formula, we get:

$$T_{n+1} = \alpha t_n + (1-\alpha)\,\alpha t_{n-1} + \ldots + (1-\alpha)^j \alpha T_{n-1} + \ldots + (1-\alpha)^{n+1} T_0$$

- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor.

End