OS2020_WilliamStallings_Homework_Chapter2

02-OS Intro (2020.9.17)

> ***Problem 2.1:***
>
> *Suppose that we have a multiprogrammed computer in which each job has identical characteristics. In one computation period, T, for a job, half the time is spent in I/O and the other half in processor activity. Each job runs for a total of N periods. Assume that a simple round-robin scheduling is used, and that I/O operations can overlap with processor operation. Define the following quantities:*
>
> - *Turnaround time=actual time to complete a job*
> - *Throughput=average number of jobs completed per time period T*
> - *Processor utilization=percentage of time that the processor is active (not waiting)*
>
> *Compute these quantities for one, two, and four simultaneous jobs, assuming that the period T is distributed in each of the following ways:*
>
> ***a.*** *I/O first half, processor second half*
>
> ***b.*** *I/O first and fourth quarters, processor second and third quarter*

Answer for Problem 2.1:

2.1 The answers are the same for **(a)** and **(b)**. Assume that although processor operations cannot overlap, I/O operations can.

Number of jobs	TAT	Throughput	Processor utilization
1	NT	1/N	50%
2	NT	2/N	100%
4	(2N – 1)T	4/(2N – 1)	100%

> ***Problem 2.2:***
>
> *An I/O-bound program is one that, if run alone, would spend more time waiting for I/O than using the processor. A processor-bound program is the opposite. Suppose a short-term scheduling algorithm favors those programs that have used little processor time in the recent past. Explain why this algorithm favors I/O-bound programs and yet does not permanently deny processor time to processor-bound programs.*

Answer for Problem 2.2:

2.2 I/O-bound programs use relatively little processor time and are therefore favored by the algorithm. However, if a processor-bound process is denied processor time for a sufficiently long period of time, the same algorithm will grant the processor to that process since it has not used the processor at all in the recent past. Therefore, a processor-bound process will not be permanently denied access.

> ***Problem 2.3:***
>
> *Contrast the scheduling policies you might use when trying to optimize a time-sharing system with those you would use to optimize a multiprogrammed batch system.*

2.3 With time sharing, the concern is turnaround time. Time-slicing is preferred because it gives all processes access to the processor over a short period of time. In a batch system, the concern is with throughput, and the less context switching, the more processing time is available for the processes. Therefore, policies that minimize context switching are favored.

> ***Problem 2.4:***
>
> *What is the purpose of system calls, and how do system calls relate to the OS and to the concept of dual-mode (kernel-mode and user-mode) operation?*

Answer for Problem 2.4:

2.4 A system call is used by an application program to invoke a function provided by the operating system. Typically, the system call results in transfer to a system program that runs in kernel mode.

> ***Problem 2.6:***
>
> *A multiprocessor with eight processors has 20 attached tape drives. There is a large number of jobs submitted to the system that each require a maximum of four tape drives to complete execution. Assume that each job starts running with only three tape drives for a long period before requiring the fourth tape drive for a short period toward the end of its operation. Also assume an endless supply of such jobs.*
>
> ***a.*** *Assume the scheduler in the OS will not start a job unless there are four tape drives available. When a job is started, four drives are assigned immediately and are not released until the job finishes. What is the maximum number of jobs that can be in progress at once? What are the maximum and minimum number of tape drives that may be left idle as a result of this policy?*
>
> ***b.*** *Suggest an alternative policy to improve tape drive utilization and at the same time avoid system deadlock. What is the maximum number of jobs that can be in progress at once? What are the bounds on the number of idling tape drives?*

Answer for Problem 2.6:

2.6 a. If a conservative policy is used, at most 20/4 = 5 processes can be active simultaneously. Because one of the drives allocated to each process can be idle most of the time, at most 5 drives will be idle at a time. In the best case, none of the drives will be idle.

b. To improve drive utilization, each process can be initially allocated with three tape drives. The fourth one will be allocated on demand. In this policy, at most $\lfloor 20/3 \rfloor = 6$ processes can be active simultaneously. The minimum number of idle drives is 0 and the maximum number is 2.

OS2020_WilliamStallings_Homework_Chapter3

03-Hardware Support (2020.10.12)

> ***Problem 1.3:***
>
> *Consider a hypothetical 32-bit microprocessor having 32-bit instructions composed of two fields. The first byte contains the opcode and the remainder an immediate operand or an operand address.*
>
> ***a.*** *What is the maximum directly addressable memory capacity (in bytes)?*
>
> ***b.*** *Discuss the impact on the system speed if the microprocessor bus has*
>
> 1. *a 32-bit local address bus and a 16-bit local data bus, or*
> 2. *a 16-bit local address bus and a 16-bit local data bus.*
>
> ***c.*** *How many bits are needed for the program counter and the instruction register?*

Answer for Problem 1.3:

1.3 a. 2^{24} = 16 MBytes

b. **(1)** If the local address bus is 32 bits, the whole address can be transferred at once and decoded in memory. However, since the data bus is only 16 bits, it will require 2 cycles to fetch a 32-bit instruction or operand.
(2) The 16 bits of the address placed on the address bus can't access the whole memory. Thus a more complex memory interface control is needed to latch the first part of the address and then the second part (since the microprocessor will end in two steps). For a 32-bit address, one may assume the first half will decode to access a "row" in memory, while the second half is sent later to access a "column" in memory. In addition to the two-step address operation, the microprocessor will need 2 cycles to fetch the 32 bit instruction/operand.

c. The program counter must be at least 24 bits. Typically, a 32-bit microprocessor will have a 32-bit external address bus and a 32-bit program counter, unless on-chip segment registers are used that may work with a smaller program counter. If the instruction register is to contain the whole instruction, it will have to be 32-bits long; if it will contain only the op code (called the op code register) then it will have to be 8 bits long.

> ***Problem 1.4:***
>
> *Consider a hypothetical microprocessor generating a 16-bit address (e.g., assume that the program counter and the address registers are 16 bits wide) and having a 16-bit data bus.*

> ***a.*** *What is the maximum memory address space that the processor can access directly if it is connected to a "16-bit memory"?*
>
> ***b.*** *What is the maximum memory address space that the processor can access directly if it is connected to an "8-bit memory"?*
>
> ***c.*** *What architectural features will allow this microprocessor to access a separate "I/O space"?*
>
> ***d.*** *If an input and an output instruction can specify an 8-bit I/O port number, how many 8-bit I/O ports can the microprocessor support? How many 16-bit I/O ports? Explain.*

Answer for Problem 1.4:

1.4 In cases **(a)** and **(b)**, the microprocessor will be able to access 2^{16} = 64K bytes; the only difference is that with an 8-bit memory each access will transfer a byte, while with a 16-bit memory an access may transfer a byte or a 16-byte word. For case **(c)**, separate input and output instructions are needed, whose execution will generate separate "I/O signals" (different from the "memory signals" generated with the execution of memory-type instructions); at a minimum, one additional output pin will be required to carry this new signal. For case **(d)**, it can support 2^8 = 256 input and 2^8 = 256 output byte ports and the same number of input and output 16-bit ports; in either case, the distinction between an input and an output port is defined by the different signal that the executed input or output instruction generated.

> ***Problem 1.7:***
>
> *In virtually all systems that include DMA modules, DMA access to main memory is given higher priority than processor access to main memory. Why?*

Answer for Problem 1.7:

1.7 If a processor is held up in attempting to read or write memory, usually no damage occurs except a slight loss of time. However, a DMA transfer may be to or from a device that is receiving or sending data in a stream (e.g., disk or tape), and cannot be stopped. Thus, if the DMA module is held up (denied continuing access to main memory), data will be lost.

> ***Problem 1.8:***
>
> *A DMA module is transferring characters to main memory from an external device transmitting at 9600 bits per second (bps). The processor can fetch instructions at the rate of 1 million instructions per second. By how much will the processor be slowed down due to the DMA activity?*

Answer for Problem 1.8:

1.8 Let us ignore data read/write operations and assume the processor only fetches instructions. Then the processor needs access to main memory once every microsecond. The DMA module is transferring characters at a rate of 1200 characters per second, or one every 833 μs. The DMA therefore "steals" every 833rd cycle. This slows down the processor approximately $\frac{1}{833} \times 100\% = 0.12\%$

> ***Problem 1.9:***
>
> *A computer consists of a CPU and an I/O device D connected to main memory M via a shared bus with a data bus width of one word. The CPU can execute a maximum of 106 instructions per second. An average instruction requires five processor cycles, three of which use the memory bus. A memory read or write operation uses one processor cycle. Suppose that the CPU is continuously executing "background" programs that require 95% of its instruction execution rate but not any I/O instructions. Assume that one processor cycle equals one bus cycle. Now suppose that very large blocks of data are to be transferred between M and D.*
>
> ***a.*** *If programmed I/O is used and each one-word I/O transfer requires the CPU to execute two instructions, estimate the maximum I/O data transfer rate, in words per second, possible through D.*
>
> ***b.*** *Estimate the same rate if DMA transfer is used.*

Answer for Problem 1.9:

1.9 a. The processor can only devote 5% of its time to I/O. Thus the maximum I/O instruction execution rate is $10^6 \times 0.05 = 50{,}000$ instructions per second. The I/O transfer rate is therefore 25,000 words/second.

b. The number of machine cycles available for DMA control is

$$10^6(0.05 \times 5 + 0.95 \times 2) = 2.15 \times 10^6$$

If we assume that the DMA module can use all of these cycles, and ignore any setup or status-checking time, then this value is the maximum I/O transfer rate.

OS2020_WilliamStallings_Homework_Chapter4

04-Process Concept (2020.10.12)

> ***Problem 3.3:***
>
> *Figure 3.9b contains seven states. In principle, one could draw a transition between any two states, for a total of 42 different transitions.*
>
> ***a.*** *List all of the possible transitions and give an example of what could cause each transition.*
>
> ***b.*** *List all of the impossible transitions and explain why.*

Answer for Problem 3.3:

3.3 a. **New → Ready or Ready/Suspend**: covered in text
Ready → Running or Ready/Suspend: covered in text
Ready/Suspend → Ready: covered in text
Blocked → Ready or Blocked/Suspend: covered in text
Blocked/Suspend → Ready /Suspend or Blocked: covered in text
Running → Ready, Ready/Suspend, or Blocked: covered in text
Any State → Exit: covered in text

b. **New → Blocked, Blocked/Suspend, or Running**: A newly created process remains in the new state until the processor is ready to take on an additional process, at which time it goes to one of the Ready states.
Ready → Blocked or Blocked/Suspend: Typically, a process that is ready cannot subsequently be blocked until it has run. Some systems may allow the OS to block a process that is currently ready, perhaps to free up resources committed to the ready process.
Ready/Suspend → Blocked or Blocked/Suspend: Same reasoning as preceding entry.
Ready/Suspend → Running: The OS first brings the process into memory, which puts it into the Ready state.
Blocked → Ready /Suspend: this transition would be done in 2 stages. A blocked process cannot at the same time be made ready and suspended, because these transitions are triggered by two different causes.

Blocked → Running: When a process is unblocked, it is put into the Ready state. The dispatcher will only choose a process from the Ready state to run
Blocked/Suspend → Ready: same reasoning as Blocked → Ready /Suspend
Blocked/Suspend → Running: same reasoning as Blocked → Running
Running → Blocked/Suspend: this transition would be done in 2 stages
Exit → Any State: Can't turn back the clock

> ***Problem 3.5:***
>
> *Consider the state transition diagram of Figure 3.9b. Suppose that it is time for the OS to dispatch a process and that there are processes in both the Ready state and the Ready/Suspend state, and that at least one process in the Ready/Suspend state has higher scheduling priority than any of the processes in the Ready state. Two extreme policies are as follows: (1) Always dispatch from a process in the Ready state, to minimize swapping, and (2) always give preference to the highest-priority process, even though that may mean swapping when swapping is not necessary. Suggest an intermediate policy that tries to balance the concerns of priority and performance*

Answer for Problem 3.5:

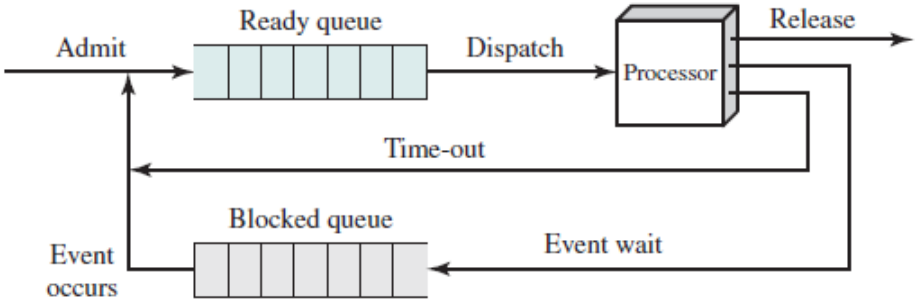
3.5 Penalize the Ready, suspend processes by some fixed amount, such as one or two priority levels, so that a Ready, suspend process is chosen next only if it has a higher priority than the highest-priority Ready process by several levels of priority.

> ***Problem 3.9:***
>
> *Figure 3.8b suggests that a process can only be in one event queue at a time.*
>
> *a. Is it possible that you would want to allow a process to wait on more than one event at the same time? Provide an example.*
>
> *b. In that case, how would you modify the queueing structure of the figure to support this new feature?*

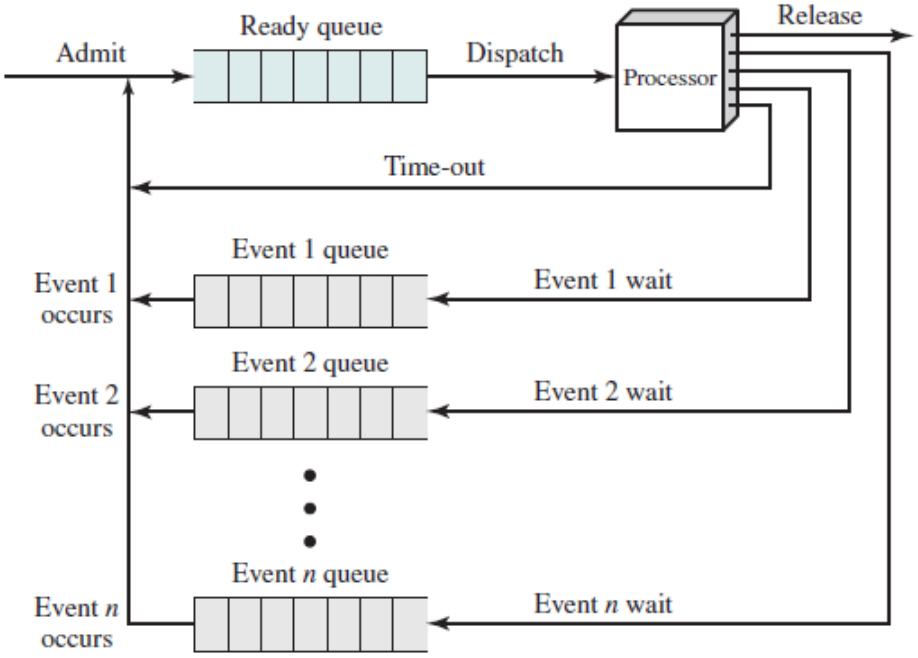
Answer for Problem 3.9:

3.9 a. An application may be processing data received from another process and storing the results on disk. If there is data waiting to be taken from the other process, the application may proceed to get that data and process it. If a previous disk write has completed and there is processed data to write out, the application may proceed to write to disk. There may be a point where the process is waiting both for additional data from the input process and for disk availability.

b. There are several ways that could be handled. A special type of either/or queue could be used. Or the process could be put in two separate queues. In either case, the operating system would have to handle the details of alerting the process to the occurrence of both events, one after the other.

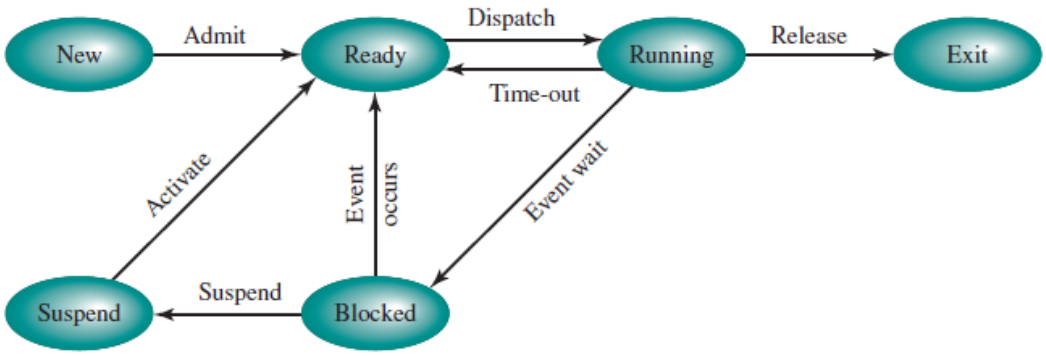


(a) Single blocked queue

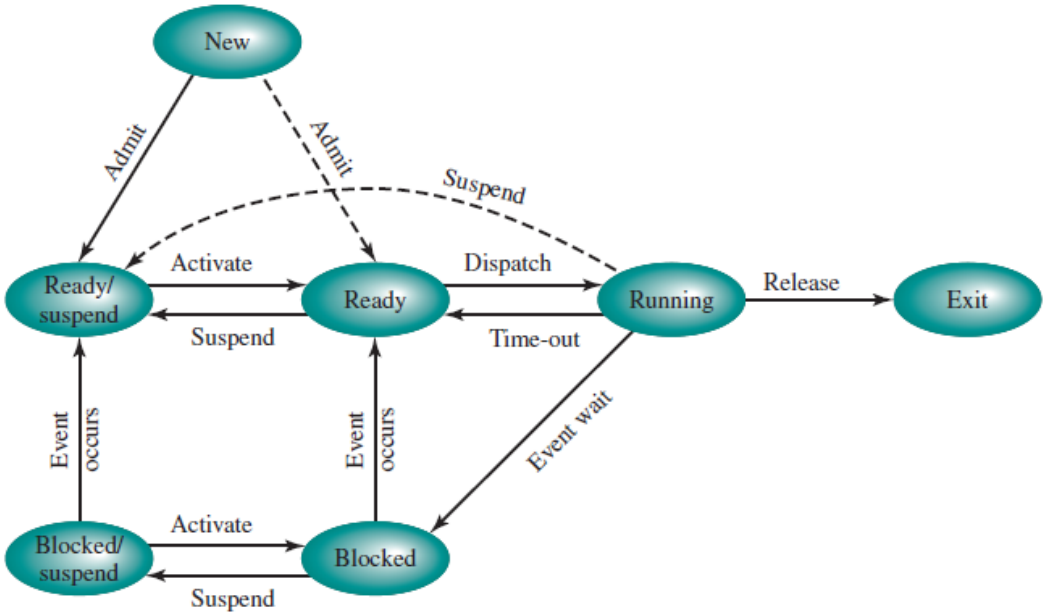


(b) Multiple blocked queues

Figure 3.8 Queueing Model for Figure 3.6



(a) With one suspend state

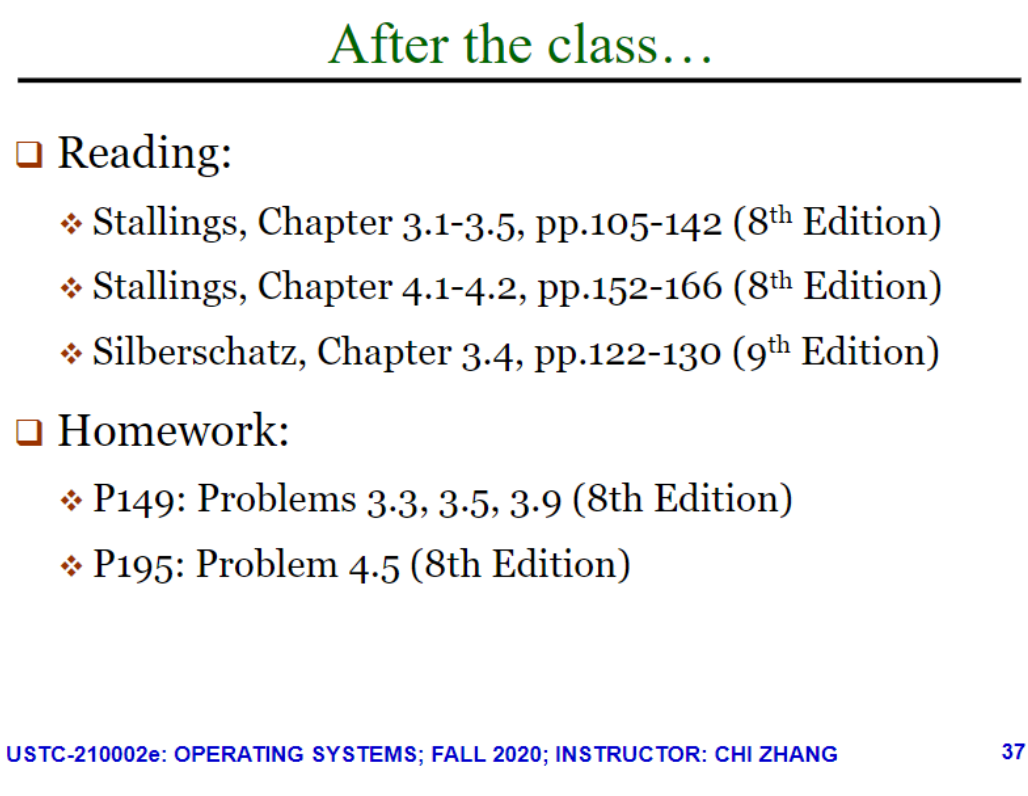


(b) With two suspend states

Figure 3.9 **Process State Transition Diagram with Suspend States**

OS2020_WilliamStallings_Homework_Chapter5

作业题目



作业内容

作业3.3、3.5、3.9已经布置过了，常见第四章作业。

> *4.5. If a process exits and there are still threads of that process running, will they continue to run?*

Answer:

No. When a process exits, it takes everything with it—the KLTs, the process structure, the memory space, everything—including threads.

OS2020_WilliamStallings_Homework_Chapter6

作业内容

After the class…

- Reading:
  - Stallings, Chapter 9.1-9.4, pp.397-428 (8th Edition)
- Homework:
  - P428: Problems 9.1, 9.2, 9.3 (8th Edition)

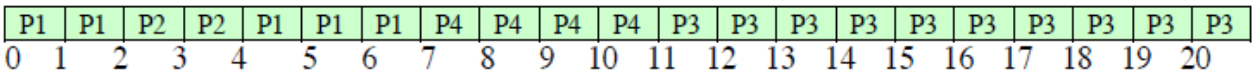
作业题目

> ***9.1***
>
> *Consider the following workload:*
>
> | Process | Burst Time | Priority | Arrival Time |
> |---|---|---|---|
> | P1 | 50 ms | 4 | 0 ms |
> | P2 | 20 ms | 1 | 20 ms |
> | P3 | 100 ms | 3 | 40 ms |
> | P4 | 40 ms | 2 | 60 ms |
>
> ***a.*** *Show the schedule using shortest remaining time, nonpreemptive priority (a smaller priority number implies higher priority) and round robin with quantum 30 ms. Use time scale diagram as shown below for the FCFS example to show the schedule for each requested scheduling policy.*
>
> *Example for FCFS (1 unit = 10 ms):*
>
> | P1 | P1 | P1 | P1 | P1 | P2 | P2 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | P3 | P4 | P4 | P4 | P4 |
> |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
>
> ***b.*** *What is the average waiting time of the above scheduling policies?*

Answer:

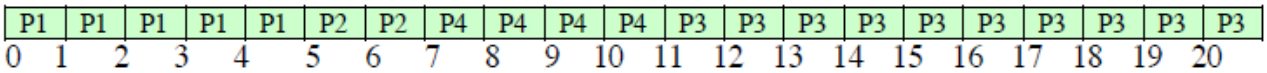
a.

(1) **Shortest Remaining Time**:



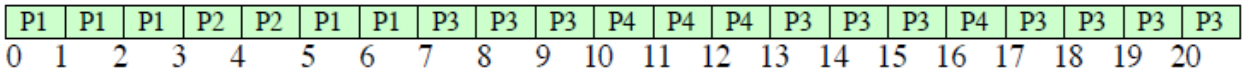
Explanation: P1 starts but is preempted after 20ms when P2 arrives and has shorter burst time (20ms) than the remaining burst time of P1 (30 ms) . So, P1 is preempted. P2 runs to completion. At 40ms P3 arrives, but it has a longer burst time than P1, so P1 will run. At 60ms P4 arrives. At this point P1 has a remaining burst time of 10 ms, which is the shortest time, so it continues to run. Once P1 finishes, P4 starts to run since it has shorter burst time than P3.

(2) **Non-preemptive Priority**:



Explanation: P1 starts, but as the scheduler is non-preemptive, it continues executing even though it has lower priority than P2. When P1 finishes, P2 and P3 have arrived. Among these two, P2 has higher priority, so P2 will be scheduled, and it keeps the processor until it finishes. Now we have P3 and P4 in the ready queue. Among these two, P4 has higher priority, so it will be scheduled. After P4 finishes, P3 is scheduled to run.

(3) **Round Robin with quantum of 30 ms**:



Explanation: P1 arrives first, so it will get the 30ms quantum. After that, P2 is in the ready queue, so P1 will be preempted and P2 is
scheduled for 20ms. While P2 is running, P3 arrives. Note that P3 will be queued after P1 in the FIFO ready queue. So when P2 is done, P1 will be scheduled for the next quantum. It runs for 20ms. In the mean time, P4 arrives and is queued after P3. So after P1 is done, P3 runs for one 30 ms quantum. Once it is done, P4 runs for a 30ms quantum. Then again P3 runs for 30 ms, and after that P4 runs for 10 ms, and after that P3 runs for 30+10ms since there is nobody left to compete with.

b.

(1) **Shortest Remaining Time**:$(20 + 0 + 70 + 10)/4 = 25ms$. Explanation: P2 does not wait, but P1 waits 20ms, P3 waits 70ms
and P4 waits 10ms.

(2) **Non-preemptive Priority**: $(0 + 30 + 10 + 70)/4 = 27.5ms$ Explanation: P1 does not wait, P2 waits 30ms until P1 finishes, P4
waits only 10ms since it arrived at 60ms and it is scheduled at 70ms. P3 waits 70ms.

(3) **Round-Robin**:$(20 + 10 + 70 + 70)/4 = 42.5ms$ Explanation: P1 waits only for P2 (for 20ms). P2 waits only 10ms until P1 finishes the quantum (it arrives at 20ms and the quantum is 30ms). P3 waits 30ms to start, then 40ms for P4 to finish. P4 waits 40ms to start and one quantum slice for P3 to finish.

9.2.

Consider the following set of processes:

Process	Arrival Time	Processing Time
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

Perform the same analysis as depicted in Table 9.5 and Figure 9.5 for this set.

Answer:

Each square represents one time unit; the number in the square refers to the currently-running process.

FCFS	A	A	A	B	B	B	B	B	C	C	D	D	D	D	D	E	E	E	E	E
RR, q = 1	A	B	A	B	C	A	B	C	B	D	B	D	E	D	E	D	E	D	E	E
RR, q = 4	A	A	A	B	B	B	B	C	C	B	D	D	D	D	E	E	E	E	D	E
SPN	A	A	A	C	C	B	B	B	B	B	D	D	D	D	D	E	E	E	E	E
SRT	A	A	A	C	C	B	B	B	B	B	D	D	D	D	D	E	E	E	E	E
HRRN	A	A	A	B	B	B	B	B	C	C	D	D	D	D	D	E	E	E	E	E
Feedback, q = 1	A	B	A	C	B	C	A	B	B	D	B	D	E	D	E	D	E	D	E	E
Feedback, q = 2^i	A	B	A	A	C	B	B	C	B	B	D	D	E	D	D	E	E	D	E	E

		A	B	C	D	E	
	T_a	0	1	3	9	12	
	T_s	3	5	2	5	5	
FCFS	T_f	3	8	10	15	20	
	T_r	3.00	7.00	7.00	6.00	8.00	6.20
	T_r/T_s	1.00	1.40	3.50	1.20	1.60	1.74
RR $q = 1$	T_f	6.00	11.00	8.00	18.00	20.00	
	T_r	6.00	10.00	5.00	9.00	8.00	7.60
	T_r/T_s	2.00	2.00	2.50	1.80	1.60	1.98
RR $q = 4$	T_f	3.00	10.00	9.00	19.00	20.00	
	T_r	3.00	9.00	6.00	10.00	8.00	7.20
	T_r/T_s	1.00	1.80	3.00	2.00	1.60	1.88
SPN	T_f	3.00	10.00	5.00	15.00	20.00	
	T_r	3.00	9.00	2.00	6.00	8.00	5.60
	T_r/T_s	1.00	1.80	1.00	1.20	1.60	1.32
SRT	T_f	3.00	10.00	5.00	15.00	20.00	
	T_r	3.00	9.00	2.00	6.00	8.00	5.60
	T_r/T_s	1.00	1.80	1.00	1.20	1.60	1.32
HRRN	T_f	3.00	8.00	10.00	15.00	20.00	
	T_r	3.00	7.00	7.00	6.00	8.00	6.20
	T_r/T_s	1.00	1.40	3.50	1.20	1.60	1.74
FB q = 1	T_f	7.00	11.00	6.00	18.00	20.00	
	T_r	7.00	10.00	3.00	9.00	8.00	7.40
	T_r/T_s	2.33	2.00	1.50	1.80	1.60	1.85
FB	T_f	4.00	10.00	8.00	18.00	20.00	
$q = 2^i$	T_r	4.00	9.00	5.00	9.00	8.00	7.00
	T_r/T_s	1.33	1.80	2.50	1.80	1.60	1.81

> ***9.3.*** *Prove that, among nonpreemptive scheduling algorithms, SPN provides the minimum average waiting time for a batch of jobs that arrive at the same time. Assume that the scheduler must always execute a task if one is available.*

Answer:

We will prove the assertion for the case in which a batch of n jobs arrive at the same time, and ignoring further arrivals. The proof can be extended to cover later arrivals. Let the service times of the jobs be $t_1 \leq t_2 \leq \ldots \leq t_n$

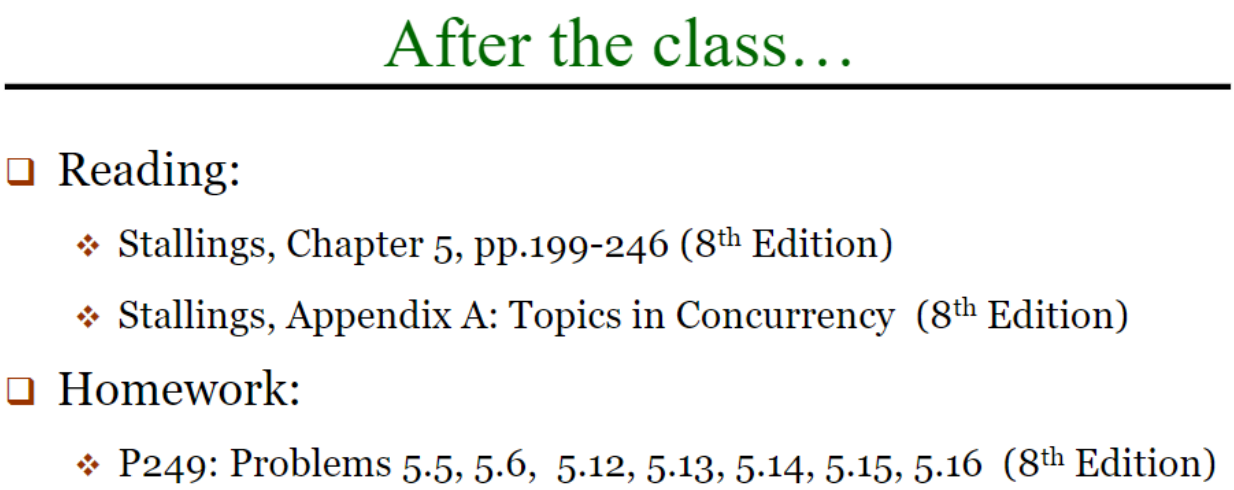
Then, n users must wait for the execution of job 1; $n-1$ users must wait for the execution of job 2, and so on. Therefore, the average response time is $\frac{n * t_1 + (n-1) * t_2 + \ldots + t_n}{n}$

If we make any changes in this schedule, for example by exchanging jobs j and k (where $j < k$), the average response time is increased by the amount $\frac{(k-j)(t_k - t_j)}{n} \geq 0$

In other words, the average response time can only increase if the SPN algorithm is not used.

OS2020_WilliamStallings_Homework_Chapter8

作业内容



作业题目

> **5.5**、*Is busy waiting always less efficient (in terms of using processor time) than a blocking wait? Explain.*

Answer:

On average, yes, because busy-waiting consumes useless instruction cycles. However, in a particular case, if a process comes to a point in the program where it must wait for a condition to be satisfied, and if that condition is already satisfied, then the busy-wait will find that out immediately, whereas, the blocking wait will consume OS resources switching out of and back into the process.

> **5.6**、*Consider the following program:*

```
boolean blocked [2];
int turn;
void P (int id)
{
    while (true) {
        blocked[id] = true;
        while (turn != id) {
            while (blocked[1-id])
```

```
            /* do nothing */;
            turn = id;
        }
        /* critical section */
        blocked[id] = false;
        /* remainder */
    }
}

void main()
{
    blocked[0] = false;
    blocked[1] = false;
    turn = 0;
    parbegin (P(0), P(1));
}
```

This software solution to the mutual exclusion problem for two processes is proposed in [HYMA66]. Find a counterexample that demonstrates that this solution is incorrect. It is interesting to note that even the Communications of the ACM was fooled on this one.

Answer:

Consider the case in which `turn` equals 0 and `P(1)` sets `blocked[1]` to `true` and then finds `blocked[0]` set to `false`. `P(0)` will then set `blocked[0]` to `true`, find `turn = 0`, and enter its critical section. `P(1)` will then assign 1 to turn and will also enter its critical section. The error was pointed out in [RAYN86].

5.12、*Consider the following definition of semaphores:*

```
void semWait(s)
{
    if (s.count > 0) {
        s.count--;
    }
    else {
        place this process in s.queue;
        block;
    }
}

void semSignal (s)
{
    if (there is at least one process blocked on semaphore s) {
        remove a process P from s.queue;
        place process P on ready list;
    }
    else
        s.count++;
}
```

> *Compare this set of definitions with that of Figure 5.3. Note one difference: With the preceding definition, a semaphore can never take on a negative value. Is there any difference in the effect of the two sets of definitions when used in programs? That is, could you substitute one set for the other without altering the meaning of the program?*

Answer:

The two are equivalent. In the definition of Figure 5.3, when the value of the semaphore is negative, its value tells you how many processes are waiting. With the definition of this problem, you don't have that information readily available. However, the two versions function the same.

> ***5.13***、*Consider a sharable resource with the following characteristics: (1) As long as there are fewer than three processes using the resource, new processes can start using it right away. (2) Once there are three process using the resource, all three must leave before any new processes can begin using it. We realize that counters are needed to keep track of how many processes are waiting and active, and that these counters are themselves shared resources that must be protected with mutual exclusion. So we might create the following solution:*

```
1   semaphore mutex = 1, block = 0;          /* share variables: semaphores, */
2   int active = 0, waiting = 0;                          /* counters, and */
3   boolean must_wait = false;                          /* state information */
4
5   semWait(mutex);                            /* Enter the mutual exclusion */
6   if(must_wait) {                          /* If there are (or were) 3, then */
7       ++waiting;                          /* we must wait, but we must leave */
8       semSignal(mutex);                      /* the mutual exclusion first */
9       semWait(block);              /* Wait for all current users to depart */
10      SemWait(mutex);                        /* Reenter the mutual exclusion */
11      --waiting;                             /* and update the waiting count */
12  }
13  ++active;                             /* Update active count, and remember */
14  must_wait = active == 3;                          /* if the count reached 3 */
15  semSignal(mutex);                        /* Leave the mutual exclusion */
16
17  /* critical section */
18
19  semWait(mutex);                                /* Enter mutual exclusion */
20  --active;                                /* and update the active count */
21  if(active == 0) {                                  /* Last one to leave? */
22      int n;
23      if (waiting < 3) n = waiting;
24      else n = 3;                                /* If so, unblock up to 3 */
25      while( n > 0 ) {                             /* waiting processes */
26          semSignal(block);
27          --n;
28      }
29  must_wait = false;                     /* All active processes have left */
30  }
31  semSignal(mutex);                          /* Leave the mutual exclusion */
```

> *The solution appears to do everything right: All accesses to the shared variables are protected by mutual exclusion, processes do not block themselves while in the mutual exclusion, new processes are prevented from using the resource if there are (or were) three active users, and the last process to depart unblocks up to three waiting processes.*
>
> ***a.*** *The program is nevertheless incorrect. Explain why.*
>
> ***b.*** *Suppose we change the if in line 6 to a while. Does this solve any problem in the program? Do any difficulties remain?*

Answer:

This problem and the next two are based on examples in; Reek, K. "Design Patterns for Semaphores." ACM SIGCSE'04, March 2004.

a. We quote the explanation in Reek's paper. There are two problems. First, because unblocked processes must reenter the mutual exclusion (line 10) there is a chance that newly arriving processes (at line 5) will beat them into the critical section. Second, there is a time delay between when the waiting processes are unblocked and when they resume execution and update the counters. The waiting processes must be accounted for as soon as they are unblocked (because they might resume execution at any time), but it may be some time before the processes actually do resume and update the counters to reflect this. To illustrate, consider the case where three processes are blocked at line 9. The last active process will unblock them (lines 25-28) as it departs. But there is no way to predict when these processes will resume executing and update the counters to reflect the fact that they have become active. If a new process reaches line 6 before the unblocked ones resume, the new one should be blocked. But the status variables have not yet been updated so the new process will gain access to the resource. When the unblocked ones eventually resume execution, they will also begin accessing the resource. The solution has failed because it has allowed four processes to access the resource together

b. This forces unblocked processes to recheck whether they can begin using the resource. But this solution is more prone to starvation
because it encourages new arrivals to "cut in line" ahead of those that were already waiting.

> *5.14、Now consider this correct solution to the preceding problem:*

```
1   semaphore mutex = 1, block = 0;          /* share variables: semaphores, */
2   int active = 0, waiting = 0;                              /* counters, and */
3   boolean must_wait = false;                          /* state information */
4
5   semWait(mutex);                            /* Enter the mutual exclusion */
6   if(must_wait) {                          /* If there are (or were) 3, then */
7       ++waiting;                          /* we must wait, but we must leave */
8       semSignal(mutex);                      /* the mutual exclusion first */
9       semWait(block);              /* Wait for all current users to depart */
10  } else {
11      ++active;                                /* Update active count, and */
12      must_wait = active == 3;          /* remember if the count reached 3 */
13      semSignal(mutex);                          /* Leave mutual exclusion */
14  }
15
16  /* critical section */
17
18  semWait(mutex);                                /* Enter mutual exclusion */
19  --active;                                    /* and update the active count */
20  if(active == 0) {                                    /* Last one to leave? */
21      int n;
22      if (waiting < 3) n = waiting;
23      else n = 3;                /* If so, see how many processes to unblock */
24      waiting -= n;                 /* Deduct this number from waiting count */
25      active = n;                           /* and set active to this number */
26      while( n > 0 ) {                       /* Now unblock the processes */
27          semSignal(block);                                  /* one by one */
28          --n;
29      }
30      must_wait = active == 3;                /* Remember if the count is 3 */
31  }
32  semSignal(mutex);                          /* Leave the mutual exclusion */
```

a. *Explain how this program works and why it is correct.*

b. *This solution does not completely prevent newly arriving processes from cutting in line but it does make it less likely. Give an example of cutting in line.*

c. *This program is an example of a general design pattern that is a uniform way to implement solutions to many concurrency problems using semaphores. It has been referred to as the I'll Do It For You pattern. Describe the pattern.*

Answer:

a. This approach is to eliminate the time delay. If the departing process updates the waiting and active counters as it unblocks waiting processes, the counters will accurately reflect the new state of the system before any new processes can get into the mutual exclusion. Because the updating is already done, the unblocked processes need not reenter the critical section at all. Implementing this pattern is easy. Identify all of the work that would have been done by an unblocked process and make the unblocking process do it instead.

b. Suppose three processes arrived when the resource was busy, but one of them lost its quantum just before blocking itself at line 9
(which is unlikely, but certainly possible). When the last active process departs, it will do three `semSignal` operations and set `must_wait` to true. If a new process arrives before the older ones resume, the new one will decide to block itself. However, it will

breeze past the `semWait` in line 9 without blocking, and when the process that lost its quantum earlier runs it will block itself instead.
This is not an error—the problem doesn't dictate which processes access the resource, only how many are allowed to access it.

c. The departing process updates the system state on behalf of the processes it unblocks.

> *5.15、Now consider another correct solution to the preceding problem:*
>
> ```
> 1   semaphore mutex = 1, block = 0;          /* share variables: semaphores, */
> 2   int active = 0, waiting = 0;                              /* counters, and */
> 3   boolean must_wait = false;                            /* state information */
> 4
>
>
>  5   semWait(mutex);                             /* Enter the mutual exclusion */
>  6   if(must_wait) {                          /* If there are (or were) 3, then */
>  7       ++waiting;                           /* we must wait, but we must leave */
>  8       semSignal(mutex);                        /* the mutual exclusion first */
>  9       semWait(block);                 /* Wait for all current users to depart */
> 10       --waiting;           /* We've got the mutual exclusion; update count */
> 11   }
> 12   ++active;                              /* Update active count, and remember */
> 13   must_wait = active == 3;                          /* if the count reached 3 */
> 14   if(waiting > 0 && !must_wait)                /* If there are others waiting */
> 15       semSignal(block);;                  /* and we don't yet have 3 active, */
> 16                                              /* unblock a waiting process */
> 17   else semSignal(mutex);               /* otherwise open the mutual exclusion */
> 18
> 19   /* critical section */
> 20
> 21   semWait(mutex);                                 /* Enter mutual exclusion */
> 22   --active;                                     /* and update the active count */
> 23   if(active == 0)                                  /* If last one to leave? */
> 24       must_wait = false;                 /* set up to let new processes enter */
> 25   if(waiting == 0 && !must_wait)               /* If there are others waiting */
> 26       semSignal(block);;                       /* and we don't have 3 active, */
> 27                                                /* unblock a waiting process */
> 28   else semSignal(mutex);               /* otherwise open the mutual exclusion */
> ```
>
> *a. Explain how this program works and why it is correct.*
>
> *b. Does this solution differ from the preceding one in terms of the number of processes that can be unblocked at a time? Explain.*
>
> *c. This program is an example of a general design pattern that is a uniform way to implement solutions to many concurrency problems using semaphores. It has been referred to as the Pass The Baton pattern. Describe the pattern.*

Answer:

a. After you unblock a waiting process, you leave the critical section (or block yourself) without opening the mutual exclusion. The unblocked process doesn't reenter the mutual exclusion—it takes over your ownership of it. The process can therefore safely update the system state on its own. When it is finished, it reopens the mutual exclusion. Newly arriving processes can no longer cut in line because they cannot enter the mutual exclusion until the unblocked process has finished. Because the unblocked process takes care of its

own updating, the cohesion of this solution is better. However, once you have unblocked a process, you must immediately stop accessing the variables protected by the mutual exclusion. The safest approach is to immediately leave (after line 26, the process leaves without opening the mutex) or block yourself.

b. Only one waiting process can be unblocked even if several are waiting—to unblock more would violate the mutual exclusion of the status variables. This problem is solved by having the newly unblocked process check whether more processes should be unblocked (line 14). If so, it passes the baton to one of them (line 15); if not, it opens up the mutual exclusion for new arrivals (line 17).

c. This pattern synchronizes processes like runners in a relay race. As each runner finishes her laps, she passes the baton to the next runner. "Having the baton" is like having permission to be on the track. In the synchronization world, being in the mutual exclusion is analogous to having the baton—only one person can have it..

> *5.16、It should be possible to implement general semaphores using binary semaphores. We can use the operations semWaitB and semSignal B and two binary semaphores, delay and mutex. Consider the following:*
>
> ```
> 1  void semWait(semaphore s)
> 2  {
> 3      semWaitB(mutex);
> 4      s--;
> 5      if (s < 0) {
> 6          semSignalB(mutex);
> 7          semWaitB(delay);
> 8      }
> 9      else SemsignalB(mutex);
> 10 }
> 11
> 12 void semSignal(semaphore s);
> 13 {
> 14     semWaitB(mutex);
> 15     s++;
> 16     if (s <= 0)
> 17     semSignalB(delay);
> 18     semSignalB(mutex);
> 19 }
> ```
>
> *Initially, s is set to the desired semaphore value. Each* `semWait` *operation decrements s, and each* `semSignal` *operation increments s. The binary semaphore* `mutex`*, which is initialized to 1, assures that there is mutual exclusion for the updating of s. The binary semaphore delay, which is initialized to 0, is used to block processes.*
>
> *There is a flaw in the preceding program. Demonstrate the flaw and propose a change that will fix it.*
>
> ***Hint:*** *Suppose two processes each call* `semWait(s)` *when s is initially 0, and after the first has just performed* `semSignalB(mutex)` *but not performed* `semWaitB(delay)`*, the second call to* `semWait(s)` *proceeds to the same point. All that you need to do is move a single line of the program.*

Answer:

Suppose two processes each call `semWait(s)` when s is initially 0, and after the first has just done `semSignalB(mutex)` but not done `semWaitB(delay)`, the second call to `semWait(s)` proceeds to the same point. Because `s = -2` and `mutex` is unlocked, if two other processes then successively execute their calls to`semSignal(s)` at that moment, they will each do `semSignalB(delay)`, but the effect of the second `semSignalB` is not defined.

The solution is to move the else line, which appears just before the end line in `semWait` to just before the end line in `semSignal`. Thus, the last `semSignalB(mutex)` in `semWait` becomes unconditional and the `semSignalB(mutex)` in `semSignal` becomes conditional. For a discussion, see "A Correct Implementation of General Semaphores," by Hemmendinger, Operating Systems Review, July 1988.

OS2020_WilliamStallings_Homework_Chapter9

作业内容

After the class…

- Reading:
  - Stallings, Chapter 6.1-6.6, pp.259-282 (8th Edition)
- Homework:
  - P302: Problems 6.5, 6.11, 6.15, 6.16, 6.18 (8th Edition)

作业题目

> *6.5、*
>
> *Given the following state for the Banker's Algorithm.*
>
> *6 processes Po through P5*
>
> *4 resource types: A (15 instances); B (6 instances) C (9 instances); D (10 instances)*
>
> *Snapshot at time To:*

Available			
A	B	C	D
6	3	5	4

Process	Current allocation A	B	C	D	Maximum demand A	B	C	D
P0	2	0	2	1	9	5	5	5
P1	0	1	1	1	2	2	3	3
P2	4	1	0	2	7	5	4	4
P3	1	0	0	1	3	3	3	2
P4	1	1	0	0	5	2	2	1
P5	1	0	1	1	4	4	4	4

a. *Verify that the Available array has been calculated correctly.*

b. *Calculate the Need matrix.*

c. *Show that the current state is safe, that is, show a safe sequence of processes. In addition, to the sequence show how the Available (working array) changes as each process terminates.*

d. *Given the request (3,2,3,3) from Process P5. Should this request be granted? Why or why not?*

Answer:

a.

$$\begin{aligned} 15-(2+0+4+1+1+1) &= 6 \\ 6-(0+1+1+0+1+0) &= 3 \\ 9-(2+1+0+0+0+1) &= 5 \\ 10-(1+1+2+1+0+1) &= 4 \end{aligned} \tag{1}$$

b. $Need\ Matrix = Max\ Matrix - Allocation\ Matrix$

process	need A	B	C	D
P0	7	5	3	4
P1	2	1	2	2
P2	3	4	4	2
P3	2	3	3	1
P4	4	1	2	1
P5	3	4	3	3

c. The following matrix shows the order in which the processes and shows what is available once the give process finishes

	available			
process	A	B	C	D
P5	7	3	6	5
P4	8	4	6	5
P3	9	4	6	6
P2	13	5	6	8
P1	13	6	7	9
P1	15	6	9	10

d. ANSWER is NO for the following reasons: IF this request were granted, then the new allocation matrix would be:

	allocation			
process	A	B	C	D
P0	2	0	2	1
P1	0	1	1	1
P2	4	1	0	2
P3	1	0	0	1
P4	1	1	0	0
P5	4	2	4	4

Then the new need matrix would be:

	allocation			
process	A	B	C	D
P0	7	5	3	4
P1	2	1	2	2
P2	3	4	4	2
P3	2	3	3	1
P4	4	1	2	1
P5	0	2	0	0

And Available is then:

Available			
A	B	C	D
3	1	2	1

Which means we could NOT satisfy ANY process' need.

> *6.11、Consider a system with a total of 150 units of memory, allocated to three processes as shown:*

Process	Max	Hold
1	70	45
2	60	40
3	60	15

Apply the banker's algorithm to determine whether it would be safe to grant each of the following requests. If yes, indicate a sequence of terminations that could be guaranteed possible. If no, show the reduction of the resulting allocation table

a. *A fourth process arrives, with a maximum memory need of 60 and an initial need of 25 units.*

b. *A fourth process arrives, with a maximum memory need of 60 and an initial need of 35 units.*

a. Creating the process would result in the state:

Process	Max	Hold	Claim	Free
1	70	45	25	25
2	60	40	20	
3	60	15	45	
4	60	25	35	

There is sufficient free memory to guarantee the termination of either P1 or P2. After that, the remaining three jobs can be
completed in any order.

b. Creating the process would result in the trivially unsafe state:

Process	Max	Hold	Claim	Free
1	70	45	25	15
2	60	40	20	
3	60	15	45	
4	60	35	25	

6.15、Consider a system consisting of four processes and a single resource. The current state of the claim and allocation matrices are:

$$\mathbf{C} = \begin{pmatrix} 3 \\ 2 \\ 9 \\ 7 \end{pmatrix} \quad \mathbf{A} = \begin{pmatrix} 1 \\ 1 \\ 3 \\ 2 \end{pmatrix}$$

What is the minimum number of units of the resource needed to be available for this state to be safe?

-

Answer:

The number of available units required for the state to be safe is 3, making a total of 10 units in the system. In the state shown in the
problem, if one additional unit is available, P2 can run to completion, releasing its resources, making 2 units available. This would allow P1 to run to completion making 3 units available. But at this point P3 needs 6 units and P4 needs 5 units. If to begin with, there had been 3 units available instead of 1 unit, there would now be 5 units available. This would allow P4 to run to completion, making 7 units available, which would allow P3 to run to completion.

> *6.16、Consider the following ways of handling deadlock: (1) banker's algorithm, (2) detect deadlock and kill thread, releasing all resources, (3) reserve all resources in advance, (4) restart thread and release all resources if thread needs to wait, (5) resource ordering, and (6) detect deadlock and roll back thread's actions.*
>
> ***a.*** *One criterion to use in evaluating different approaches to deadlock is which approach permits the greatest concurrency. In other words, which approach allows the most threads to make progress without waiting when there is no deadlock? Give a rank order from 1 to 6 for each of the ways of handling deadlock just listed, where 1 allows the greatest degree of concurrency. Comment on your ordering.*
>
> ***b.*** *Another criterion is efficiency; in other words, which requires the least processor overhead. Rank order the approaches from 1 to 6, with 1 being the most efficient, assuming that deadlock is a very rare event. Comment on your ordering. Does your ordering change if deadlocks occur frequently?*

Answer:

a. In order from most-concurrent to least, there is a rough partial order on the deadlock-handling algorithms:

1、detect deadlock and kill thread, releasing its resources

detect deadlock and roll back thread's actions,

restart thread and release all resources if thread needs to wait.

None of these algorithms limit concurrency before deadlock occurs, because they rely on runtime checks rather than static restrictions. Their effects after deadlock is detected are harder to characterize: they still allow lots of concurrency (in some cases they enhance it), but the computation may no longer be sensible or efficient. The third algorithm is the strangest, since so much of its concurrency will be useless repetition; because threads compete for execution time, this algorithm also prevents useful computation from advancing. Hence it is listed twice in this ordering, at both extremes.

2、banker's algorithm,

resource ordering

These algorithms cause more unnecessary waiting than the previous two by restricting the range of allowable computations. The banker's algorithm prevents unsafe allocations (a proper superset of deadlock-producing allocations) and resource ordering restricts allocation sequences so that threads have fewer options as to whether they must wait or not.

3、reserve all resources in advance

This algorithm allows less concurrency than the previous two, but is less pathological than the worst one. By reserving all resources in advance, threads have to wait longer and are more likely to block other threads while they work, so the system-wide execution is in effect more linear.

4、restart thread and release all resources if thread needs to wait

As noted above, this algorithm has the dubious distinction of allowing both the most and the least amount of concurrency, depending on the definition of concurrency

b. In order from most-efficient to least, there is a rough partial order on the deadlock-handling algorithms:

1、reserve all resources in advance,

resource ordering

These algorithms are most efficient because they involve no runtime overhead. Notice that this is a result of the same static
restrictions that made these rank poorly in concurrency.

2、banker's algorithm

detect deadlock and kill thread, releasing its resources

These algorithms involve runtime checks on allocations which are roughly equivalent; the banker's algorithm performs a search to verify safety which is O(n m) in the number of threads and allocations, and deadlock detection performs a cycle-detection search which is O(n) in the length of resource-dependency chains. Resource-dependency chains are bounded by the number of threads, the number of resources, and the number of allocations.

3、detect deadlock and roll back thread's actions

This algorithm performs the same runtime check discussed previously but also entails a logging cost which is O(n) in the total number of memory writes performed.

4、restart thread and release all resources if thread needs to wait

This algorithm is grossly inefficient for two reasons. First, because threads run the risk of restarting, they have a low probability of completing. Second, they are competing with other restarting threads for finite execution time, so the entire system advances towards completion slowly if at all.

This ordering does not change when deadlock is more likely. The algorithms in the first group incur no additional runtime penalty
because they statically disallow deadlock-producing execution. The second group incurs a minimal, bounded penalty when deadlock
occurs. The algorithm in the third tier incurs the unrolling cost, which is $O(n)$ in the number of memory writes performed between
checkpoints. The status of the final algorithm is questionable because the algorithm does not allow deadlock to occur; it might be the case that unrolling becomes more expensive, but the behavior of this restart algorithm is so variable that accurate comparative analysis is nearly impossible.

> *6.18、Suppose that there are two types of philosophers. One type always picks up his left fork first (a "lefty"), and the other type always picks up his right fork first (a "righty"). The behavior of a lefty is defined in Figure 6.12. The behavior of a righty is as follows:*
>
> ```
> begin
>     repeat
>         think;
>         wait ( fork[ (i+1) mod 5] );
>         wait ( fork[i] );
>         eat;
>         signal ( fork[i] );
>         signal ( fork[ (i+1) mod 5] );
>     forever
> end;
> ```
>
> *Prove the following:*
>
> ***a.*** *Any seating arrangement of lefties and righties with at least one of each avoids deadlock.*
>
> ***b.*** *Any seating arrangement of lefties and righties with at least one of each prevents starvation.*

Answer:

a. Assume that the table is in deadlock, i.e., there is a nonempty set D of philosophers such that each P_i in D holds one fork and waits for a fork held by neighbor. Without loss of generality, assume that $P_j \in D$ is a lefty. Since P_j clutches his left fork and cannot have his right fork, his right neighbor P_k never completes his dinner and is also a lefty. Therefore, $Pk \in D$. Continuing the argument rightward around the table shows that all philosophers in D are lefties. This contradicts the existence of at least one righty. Therefore deadlock is not possible.

b. Assume that lefty P_j starves, i.e., there is a stable pattern of dining in which P_j never eats. Suppose Pj holds no fork. Then P_j's left neighbor P_i must continually hold his right fork and never finishes eating. Thus P_i is a righty holding his right fork, but never getting his left fork to complete a meal, i.e., P_i also starves. Now P_i's left neighbor must be a righty who continually holds his right fork. Proceeding leftward around the table with this argument shows that all philosophers are (starving) righties. But P_j is a lefty: a contradiction. Thus P_j must hold one fork.

As P_j continually holds one fork and waits for his right fork, P_j's right neighbor Pk never sets his left fork down and never completes a meal, i.e., P_k is also a lefty who starves. If P_k did not continually hold his left fork, $P_j$could eat; therefore P_k holds his left fork.

Carrying the argument rightward around the table shows that all philosophers are (starving) lefties: a contradiction. Starvation is thus precluded.

OS2020_WilliamStallings_Homework_Chapter 10

After the class…

- Reading:
  - Stallings, Chapter 1.5-1.7, pp.24-31 (8th Edition)
  - Stallings, Chapter 7.A, pp.333-339 (8th Edition)
  - Stallings, Chapter 7.1-7.2, pp.310-324 (8th Edition)

- Homework:
  - P331: Problems 7.5, 7.6, 7.8, 7.9 (8th Edition)

> *7.5、Another placement algorithm for dynamic partitioning is referred to as worst-fit. In this case, the largest free block of memory is used for bringing in a process.*
> *a. Discuss the pros and cons of this method compared to first-, next-, and best-fit.*
> *b. What is the average length of the search for worst-fit?*

Answer:

a. A criticism of the best-fit algorithm is that the space remaining after allocating a block of the required size is so small that in general it is of no real use. The worst fit algorithm maximizes the chance that the free space left after a placement will be large enough to satisfy another request, thus minimizing the frequency of compaction. The disadvantage of this approach is that the largest blocks are allocated first; therefore a request for a large area is more likely to fail.

b. Same as best fit.

> *7.6、This diagram shows an example of memory configuration under dynamic partitioning, after a number of placement and swapping-out operations have been carried out. Addresses go from left to right; gray areas indicate blocks occupied by processes; white areas indicate free memory blocks. The last process placed is 2-Mbyte and is marked with an X. Only one process was swapped out after that.*

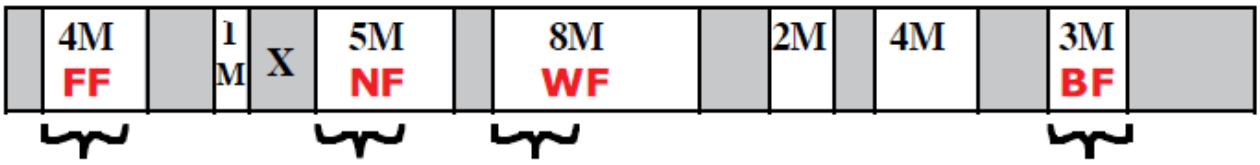
a. What was the maximum size of the swapped-out process?
b. What was the size of the free block just before it was partitioned by X?
c. A new 3-Mbyte allocation request must be satisfied next. Indicate the intervals of memory where a partition will be created for the new process under the following four placement algorithms: best-fit, first-fit, next-fit, and worst-fit. For each algorithm, draw a horizontal segment under the memory strip and label it clearly.

Answer:

a. When the 2-MB process is placed, it fills the leftmost portion of the free block selected for placement. Because the diagram shows an empty block to the left of X, the process swapped out after X was placed must have created that empty block. Therefore, the maximum size of the swapped out process is 1M.

b. The free block consisted of the 5M still empty plus the space occupied by X, for a total of 7M.

c. The answers are indicated in the following figure:



7.8、Consider a buddy system in which a particular block under the current allocation has an address of 011011110000.
a. If the block is of size 4, what is the binary address of its buddy?
b. If the block is of size 16, what is the binary address of its buddy?

Answer:

a. 011011110100

b. 011011100000

7.9、Let $buddy_k(x)$ = address of the buddy of the block of size 2k whose address is x. Write a general expression for $buddy_k(x)$.

$$\textbf{7.9}\quad \text{buddy}_k(x) = \begin{cases} x + 2^k & \text{if } x \bmod 2^{k+1} = 0 \\ x - 2^k & \text{if } x \bmod 2^{k+1} = 2^k \end{cases}$$

OS2020_WilliamStallings_Homework_Chapter 11

After the class…

- Reading:
  - Stallings, Chapter 7.3-7.7, pp.325-332 (8th Edition)
  - Stallings, Chapter 8.1-8.2, pp.340-376 (8th Edition)

- Homework:
  - P332: Problems 7.12, 7.14, 7.15 (8th Edition)
  - P392: Problems 8.1, 8.4, 8.5 (8th Edition)

> *7.12、Consider a simple paging system with the following parameters: 232 bytes of physical memory; page size of 210 bytes; 216 pages of logical address space.*
> *a. How many bits are in a logical address?*
> *b. How many bytes in a frame?*
> *c. How many bits in the physical address specify the frame?*
> *d. How many entries in the page table?*
> *e. How many bits in each page table entry? Assume each page table entry contains a valid/invalid bit.*

Answer:

a. The number of bytes in the logical address space is (2^{16} pages) × (2^{10} bytes/page) = 2^{26} bytes. Therefore, 26 bits are required for the logical address.

b. A frame is the same size as a page, 2^{10} bytes.

c. The number of frames in main memory is (2^{32} bytes of main memory)/(2^{10} bytes/frame) = 2^{22} frames. So 22 bits is needed to specify the frame.

d. There is one entry for each page in the logical address space. Therefore there are 2^{16} entries.

e. In addition to the valid/invalid bit, 22 bits are needed to to specify the frame location in main memory, for a total of 23 bits.

7.14、Consider a simple segmentation system that has the following segment table:

Starting Address	Length (bytes)
660	248
1,752	422
222	198
996	604

For each of the following logical addresses, determine the physical address or indicate if a segment fault occurs:

a. 0, 198
b. 2, 156
c. 1, 530
d. 3, 444
e. 0, 222

Answer:

a. Segment 0 starts at location 660. With the offset, we have a physical address of 660 + 198 = 858

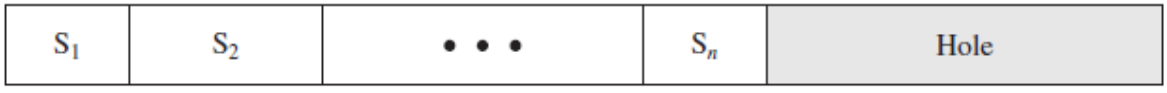
b. 222 + 156 = 378

c. Segment 1 has a length of 422 bytes, so this address triggers a segment fault.

d. 996 + 444 = 1440

e. 660 + 222 = 882

7.15、Consider a memory in which contiguous segments $S_1, S_2, \ldots, S_n$ are placed in their order of creation from one end of the store to the other, as suggested by the following figure:



When segment S_{n+1} is being created, it is placed immediately after segment Sn even though some of the segments $S_1, S_2, \ldots, S_n$ may already have been deleted. When the boundary between segments (in use or deleted) and the hole reaches the other end of the memory, the segments in use are compacted.

a. Show that the fraction of time F spent on compacting obeys the following inequality:

$$F \geq \frac{1-f}{1+kf}, where\ k = \frac{t}{2s} - 1 \quad (1)$$

where
s = average length of a segment, in words
t = average lifetime of a segment, in memory references
f = fraction of the memory that is unused under equilibrium conditions

Hint: Find the average speed at which the boundary crosses the memory and assume that the copying of a single word requires at least two memory references.

Answer:

a. Observe that a reference occurs to some segment in memory each time unit, and that one segment is deleted every t references.
Because the system is in equilibrium, a new segment must be inserted every t references; therefore, the rate of the boundary's movement is s/t words per unit time. The system's operation time t_0 is then the time required for the boundary to cross the hole, i.e., $t_0 = fmr/s$, where m = size of memory. The compaction operation requires two memory references—a fetch and a store—plus overhead for each of the $(1-f)m$ words to be moved, i.e., the compaction time tc is at least $2(1-f)m$. The fraction F of the time spent compacting is $F = 1-t_0/(t_0 + t_c)$, which reduces to the expression given.

b. $k = (t/2s)-1 = 9; F \geq (1-0.2)/(1 + 1.8) = 0.29$

8.1、Suppose the page table for the process currently executing on the processor looks like the following. All numbers are decimal, everything is numbered starting from zero, and all addresses are memory byte addresses. The page size is 1,024 bytes.

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

a. Describe exactly how, in general, a virtual address generated by the CPU is translated into a physical main memory address.
b. What physical address, if any, would each of the following virtual addresses correspond to? (Do not try to handle any page faults, if any.)
(i) 1,052
(ii) 2,221
(iii) 5,499

a. Split binary address into virtual page number and offset; use VPN as index into page table; extract page frame number; concatenate offset to get physical memory address

b.

(i) 1052 = 1024 + 28 maps to VPN 1 in PFN 7, (7 × 1024+28 =7196)
(ii) 2221 = 2 × 1024 + 173 maps to VPN 2, page fault
(iii) 5499 = 5 × 1024 + 379 maps to VPN 5 in PFN 0, (0 × 1024+379 = 379)

> *8.4、Consider the following string of page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Complete a figure similar to Figure 8.14, showing the frame allocation for:*
> *a. FIFO (first-in-first-out)*
> *b. LRU (least recently used)*
> *c. Clock*
> *d. Optimal (assume the page reference string continues with 1, 2, 0, 1, 7, 0, 1)*
> *e. List the total number of page faults and the miss rate for each policy. Count page faults only after all frames have been initialized.*

Answer:

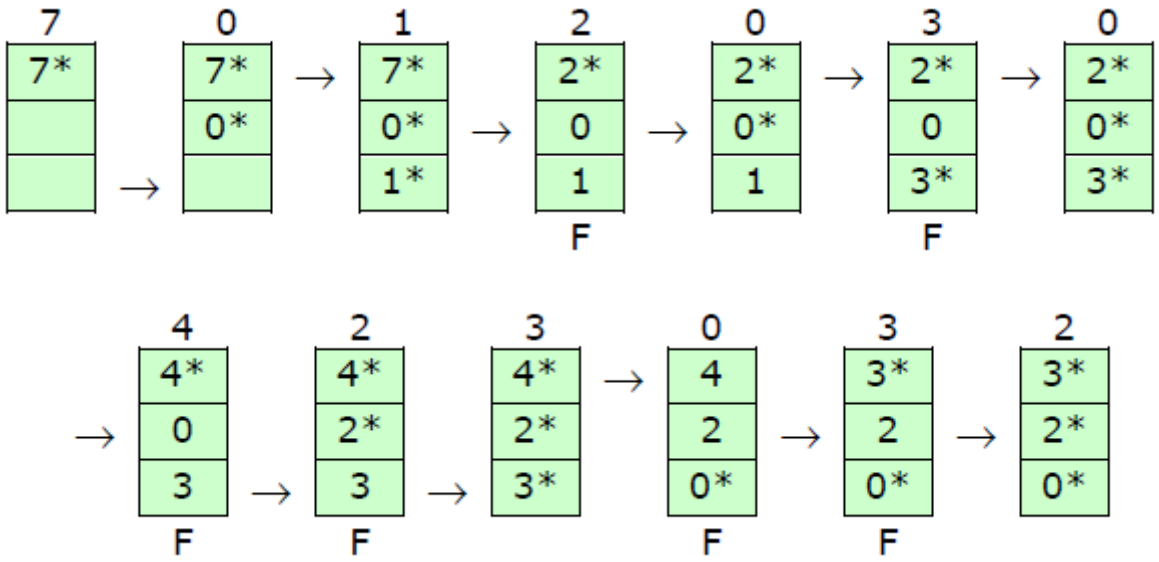
a. FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2
7	7	7	2	2	2	2	4	4	4	0	0	0
	0	0	0	0	3	3	3	2	2	2	2	2
		1	1	1	1	0	0	0	3	3	3	3
			F		F	F	F	F	F	F		

b. LRU

7	0	1	2	0	3	0	4	2	3	0	3	2
7	7	7	2	2	2	2	4	4	4	0	0	0
	0	0	0	0	0	0	0	0	3	3	3	3
		1	1	1	3	3	3	2	2	2	2	2
			F		F		F	F	F	F		

c. Clock



d. OPT

7	0	1	2	0	3	0	4	2	3	0	3	2
7	7	7	2	2	2	2	2	2	2	2	2	2
	0	0	0	0	0	0	4	4	4	0	0	0
		1	1	3	3	3	3	3	3	3	3	3
			F		F					F		

e.

FIFO: page faults = 7 miss rate = 70%
LRU: page faults = 6 miss rate = 60%
Clock: page faults = 6 miss rate = 60%
OPT: page faults = 3 miss rate = 30%

> *8.5、A process references five pages, A, B, C, D, and E, in the following order:*
> *A; B; C; D; A; B; E; A; B; C; D; E*
> *Assume that the replacement algorithm is first-in-first-out and find the number of page transfers during this sequence of references starting with an empty main memory with three page frames. Repeat for four page frames.*

Answer:

9 and 10 page transfers, respectively. This is referred to as "Belady's anomaly," and was reported in "An Anomaly in Space-Time
Characteristics of Certain Programs Running in a Paging Machine," by Belady et al, Communications of the ACM, June 1969.

OS2020_WilliamStallings_Homework_Chapter 12

After the class…

- Reading:
  - Stallings, Chapter 1.7, pp.31-33 (8th Edition)
  - Stallings, Chapter 1.4, pp.13-23 (8th Edition)
  - Stallings, Chapter 11.1-11.7, pp.477-508 (8th Edition)
  - Tanenbaum, Modern Operating Systems, Chapter 5.1-5.3, pp.336-369 (4th Edition)
- Homework:
  - P520: Problems 11.1, 11.2, 11.3 (8th Edition)

> *11.1、Consider a program that accesses a single I/O device and compare unbuffered I/O to the use of a buffer. Show that the use of the buffer can reduce the running time by at most a factor of two.*

Answer:

If the calculation time exactly equals the I/O time (which is the most favorable situation), both the processor and the peripheral device running simultaneously will take half as long as if they ran separately. Formally, let C be the calculation time for the entire program and let T be the total I/O time required. Then the best possible running time with buffering is $\max(C, T)$, while the running time without buffering is $C + T$; and of course $((C + T)/2) \leq \max(C, T) \leq (C + T)$.

Source: [KNUT97].

> *11.2、Generalize the result of Problem 11.1 to the case in which a program refers to n devices.*

Answer:

The best ratio is $(n+1) : n$.

Source: [KNUT97].

> *11.3*、
>
> *a. Perform the same type of analysis as that of Table 11.2 for the following sequence of disk track requests: 27, 129, 110, 186, 147, 41, 10, 64, 120. Assume that the disk head is initially positioned over track 100 and is moving in the direction of decreasing track number.*
>
> *b. Do the same analysis, but now assume that the disk head is moving in the direction of increasing track number.*

Answer:

a. Disk head is initially moving in the direction of decreasing track number:

FIFO		SSTF		SCAN		C-SCAN	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
27	73	110	10	64	36	64	36
129	102	120	10	41	23	41	23
110	19	129	9	27	14	27	14
186	76	147	18	10	17	10	17
147	39	186	39	110	100	186	176
41	106	64	122	120	10	147	39
10	31	41	23	129	9	129	18
64	54	27	14	147	18	120	9
120	56	10	17	186	39	110	10
Average	61.8	Average	29.1	Average	29.6	Average	38

b. If the disk head is initially moving in the direction of increasing track number, only the SCAN and C-SCAN results change:

SCAN		C-SCAN	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
110	10	110	10
120	10	120	10
129	9	129	9
147	18	147	18
186	39	186	39
64	122	10	176
41	23	27	17
27	14	41	14
10	17	64	23
Average	29.1	Average	35.1

OS2020_WilliamStallings_Homework_Chapter 13

After the class…

- Reading:
  - Stallings, Chapter 12.1-12.7, pp.522-552 (8th Edition)
  - Tanenbaum, Modern Operating Systems, Chapter 4.1-4.4, pp.263-320 (4th Edition)

- Homework:
  - P569: Problems 12.1, 12.2, 12.3, 12.13 (8th Edition)

> *12.1*、
>
> *Define:*
> B = *block size*
> R = *record size*
> P = *size of block pointer*
> F = *blocking factor; expected number of records within a block*
> *Give a formula for* F *for the three blocking methods depicted in Figure 12.8.*

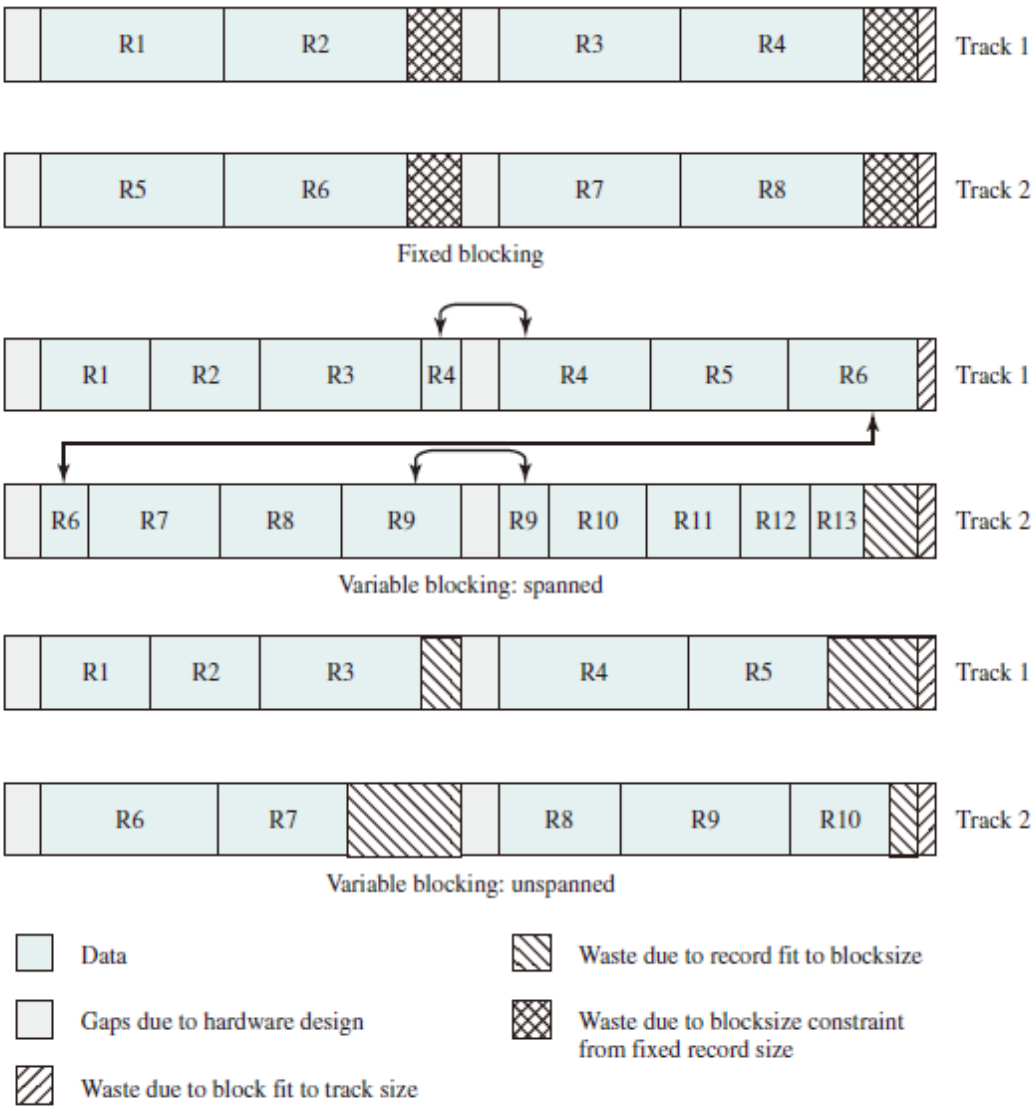


Figure 12.8 Record Blocking Methods [WIED87]

Answer:

Fixed blocking: $F = \text{largest integer} \leq \frac{B}{R}$

When records of variable length are packed into blocks, data for marking the record boundaries within the block has to be added to separate the records. When spanned records bridge block boundaries, some reference to the successor block is also needed. One possibility is a length indicator preceding each record. Another possibility is a special separator marker between records. In any case, we can assume that each record requires a marker, and we assume that the size of a marker is about equal to the size of a block pointer [WEID87]. For spanned blocking, a block pointer of size P to its successor block may be included in each block, so that the pieces of a spanned record can easily be retrieved. Then we have

Variable-length spanned blocking: $F = \frac{B - P}{R + P}$

With unspanned variable-length blocking, an average of $R/2$ will be wasted because of the fitting problem, but no successor pointer is required:

Variable-length unspanned blocking: $f = \dfrac{B - \dfrac{R}{2}}{R}$

> *12.2、One scheme to avoid the problem of preallocation versus waste or lack of contiguity is to allocate portions of increasing size as the file grows. For example, begin with a portion size of one block, and double the portion size for each allocation. Consider a file of n records with a blocking factor of F, and suppose that a simple one-level index is used as a file allocation table.*
> ***a.*** *Give an upper limit on the number of entries in the file allocation table as a function of F and n.*
> ***b.*** *What is the maximum amount of the allocated file space that is unused at any time?*

Answer:

a. $log_2 \dfrac{N}{F}$

b. Less than half the allocated file space is unused at any time.

> *12.3、What file organization would you choose to maximize efficiency in terms of speed of access, use of storage space, and ease of updating (adding/deleting/modifying) when the data are*
>
> ***a.*** *updated infrequently and accessed frequently in random order?*
> ***b.*** *updated frequently and accessed in its entirety relatively frequently?*
> ***c.*** *updated frequently and accessed frequently in random order?*

Answer:

a. Indexed

b. Indexed sequential

c. Hashed or indexed

> *12.13、Consider the organization of a UNIX file as represented by the inode (Figure 12.15). Assume that there are 12 direct block pointers, and a singly, doubly, and triply indirect pointer in each inode. Further, assume that the system block size and the disk sector size are both 8K. If the disk block pointer is 32 bits, with 8 bits to identify the physical disk and 24 bits to identify the physical block, then*
> ***a.*** *What is the maximum file size supported by this system?*
> ***b.*** *What is the maximum file system partition supported by this system?*
> ***c.*** *Assuming no information other than that the file inode is already in main memory, how many disk accesses are required to access the byte in position 13,423,956?*

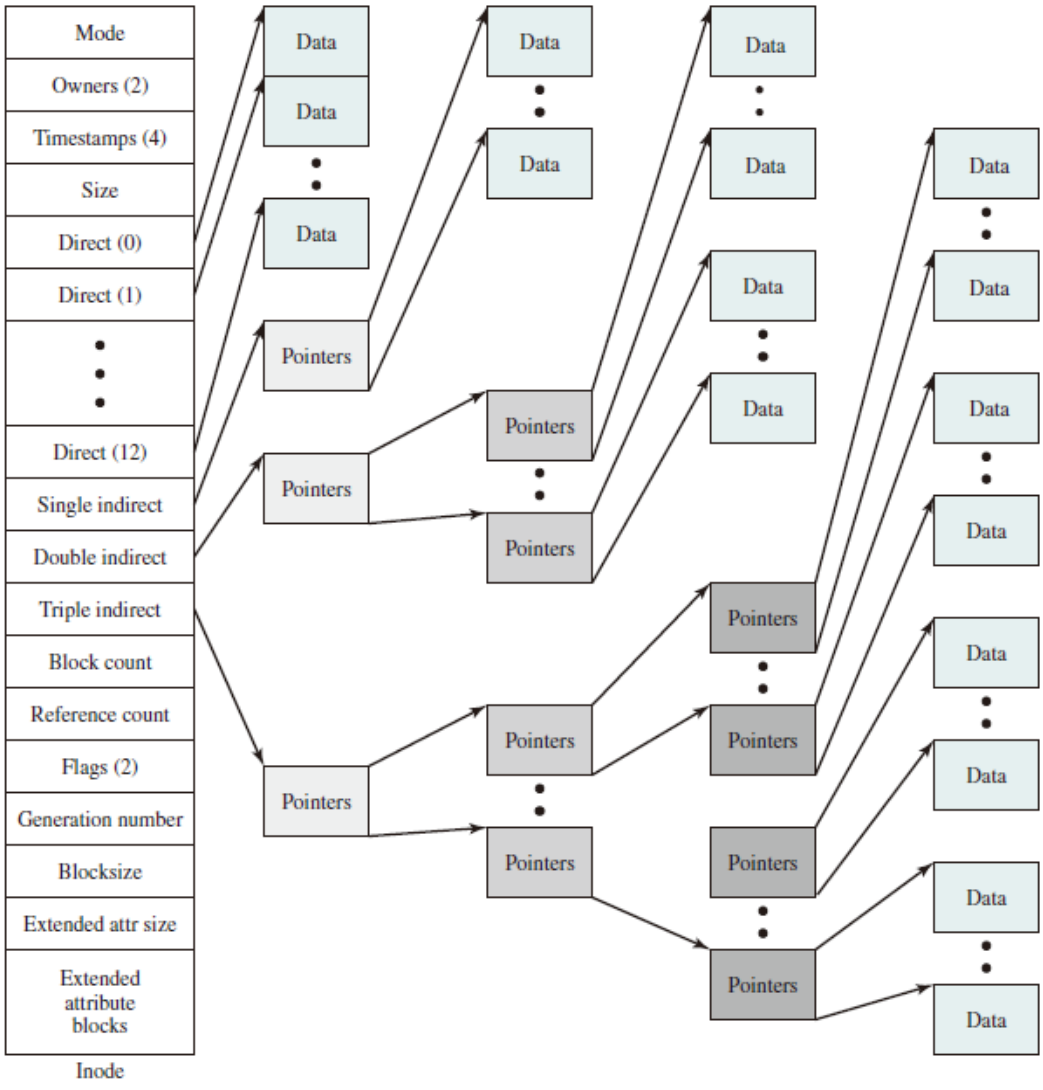


Figure 12.15 Structure of FreeBSD Inode and File

Answer:

a. Find the number of disk block pointers that fit in one block by dividing the block size by the pointer size:

$$8K/4 = 2K \text{ pointers per block}$$

The maximum file size supported by the inode is thus:

12	+ 2K	+ (2K × 2K)	+ (2K × 2K × 2K)
Direct	Indirect – 1	Indirect – 2	Indirect – 3
12	+ 2K	+ 4M	+ 8G blocks

Which, when multiplied by the block size $(8K)$, is

$$96KB + 16MB + 32GB + 64TB$$

Which is HUGE.

b. There are 24 bits for identifying blocks within a partition, so that leads to:

$$2^{24} \times 8K = 16M \times 8K = 128\ GB$$

c. Using the information from (a), we see that the direct blocks only cover the first $96KB$, while the first indirect block covers the next $16MB$. the requested file position is $13M$ and change, which clearly falls within the range of the first indirect block. There will thus be two disk accesses. One for the first indirect block, and one for the block containing the required data.