Lecture 10

04.13.2023

Midterm:solution

Q1: Infinite Buffer Producer/Consumer Problem

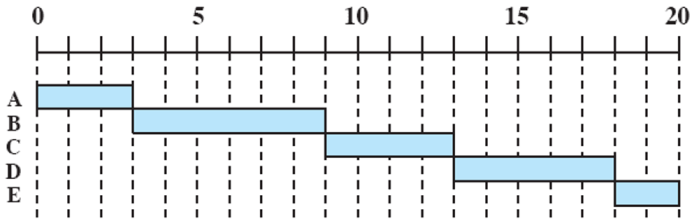
Directly taken from the Lecture Notes

Q2: Directly taken from Lecture Notes + PPT.

Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

First-Come-First Served (FCFS)



Q3:

Table 9.3 Characteristics of Various Scheduling Policies

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s − e]	$\max\left(\frac{w+s}{s}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

Q4:

```
do{
  while (lock){NULL}
  lock = TRUE;
  critical section;
  lock = FALSE;
  reminder section;
} while(TRUE);
```

```
do{
  while(lock){NULL}
  lock = TRUE;
  critical section;
  lock = FALSE;
  reminder section;
} while(TRUE);
```

Which of the following statements is true regarding the proposed algorithm

a) Mutual exclusion to the critical section is guaranteed

b) Both processes can be in their critical section at the same time

c) Lock should be initialized to TRUE

d) None of the above

> Reason: It's all depends on the value of lock. If lock is true, both them will spin forever.While lock is true is equal to true spend forever. If lock is false, both of them will be enter the critical section.
>
> c: If True, both of them will spin forever.

Q5: Homework

part1: tally =2, tally = 100;

tally =3, tally = 150;

Q6: Semaphores are atomic or non-atomic operation.

Question is whether if they are not atomic will still work and the answer is no. Because both of them could enter the critical section.

If they both execute the semaphore and the value of s count is greater than 0. If is not then they can access it at the same time.

Q7: Taken verbatim from the homework

a) Illegeal

b) legal

c) illegal

Chapter 8

8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7

8.8, 8.9 Read

Chapter 9

9.1, 9.2, 9.3, 9.4

Read: 9.5, 9.6, 9.7, 9.8

DeadLocks and Starvation

Processes **Resources**

More Processes **Than** **we satisfy with Resources.**

Processes **Resources**

too many resources demands **scarce**

Processes resources: Concurrency

recipe result in trouble.

That's in a nutshell what goes inside the OS. And OS has to address this problem.

Processes fight for CPU: we addressed that.

Processes fight for accessing a shared resources (Critical section) we addressed that .

Processes fight to have exclusive access to common resources in a concurrent environment and that we must address now. If not processes will end up in a deadlocked scenario in which no one is able to proceed.

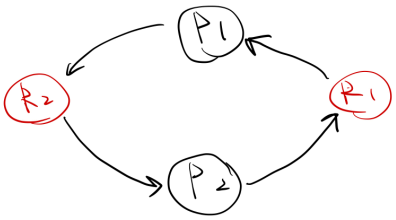
Simplest Example

Two processes P1 and P2

Two resources (Files) R1, R2

Each P1 and P2 must have R1 and R2 in order to proceed. P1 has acquired R1 and P2 has acquired R2

Both P1 and P2 are deadlocked.



Concurrency. OS of 21 centuries.

No concurrency is the OS 1960-1970s ERA.

OS has to intervene must address the problem of Deadlock and Starvation.

Three ways:

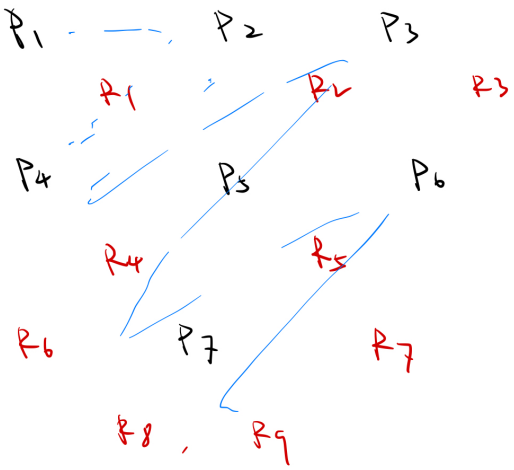
1. Deadlock Prevention

   Process
2. Deadlock Avoidance

   Future
3. Deadlock Detection

   Past

None of these solutions are optimal.



They are all accessing each other computing for fighting among each other. Right here, something happening inside the computer. There are all interacting taking files storing files. Opening accessing USBS and so forth.

Conditions is Deadlock likely to take place.

Necessary + sufficient condition for Deadlock to happen.

Necessary + sufficient condition for Deadlock

$x \equiv y \rightarrow xy + x'y'$

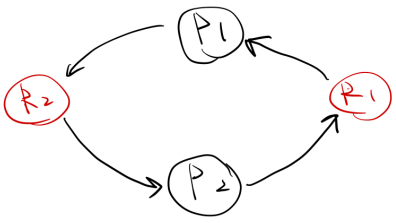
This is an equivalence relationship. Says x condition is equivalent to y, which means this if you want to do it to a predicate logic either x and y must happen, or not x and not y must happen.

In simple English for a deadlock to happen all conditions must be true. If any one of the conditions is false, deadlock is not going to happen. Break one single conditions and you are done.

The next logical Question:

What are these conditions that if we break one of them, we will happy (No Deadlock)

1. Muted Exclusion: Only one process may use a resource at a time.
2. Hold and wait: A process may hold resources it acquired indefinitely
3. No-preemption: No process can be forced to relinquish its resources.
4. Circular wait: It's a closed chain of processes such that a process holds at least one resources need by the next process in the chain.



These four conditions must exist in the same in order for Deadlock.

Break one of them means no deadlock.

Three strategies to deal with deadlocks:

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection

Deadlock Prevention

- Break one of the conditions.

Deadlock Avoidance

- Allow the first three conditions to occur, but make a careful decision so that you prevent the condition of circular wait to happen. That is when a process makes a new resource request, make sure that the resources request if granted may not lead to a dead lock.

  New request -> No deadlock -> safe state

  New request -> might deadlock -> unsafe state.

  Unsafe state does not imply that for sure will deadlock. Pessimistic

From PPT

Deadlock Avoidance

- A decision is made dynamically whether the current resources allocation request will, if granted, potentially lead to a deadlock.
- Requires knowledge of future process requests

Deadlock avoidance allows the three necessary conditions, but makes judicious choices to assure that the deadlock point is never reached. Avoidance allows more concurrency than prevention.

Two Approaches to Deadlock Avoidance

- Process Initiation Denial
  - Do not start a process if its demands might lead to deadlock
- Resource Allocation Denial
  - Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Process Initiation Denial

- A process is only started if the maximum claim of all current processes plus those of the new process can be met.
- Not optimal
  - Assumes the worst: that all processes will make their maximum claims together.

Allocation Denial

- Referred to as the banker's algorithm
  - A strategy of resource allocation denial
- Consider a system with fixed number of resources
  - **State** of the system is the current allocation of resources to process

- **Safe state** is where there is at least one sequence that does not result in deadlock
- **Unsafe state** is a state that is not safe

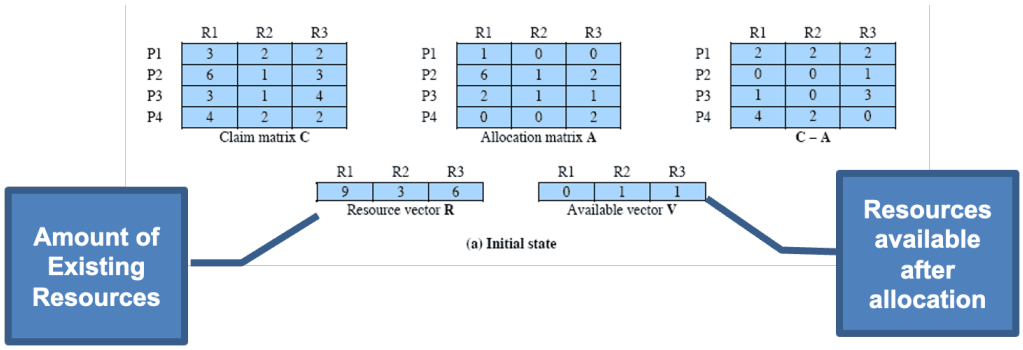
Final Exam

Determination of Safe State

- **A system consisting of four processes and three resources.**
- **Allocations are made to processors**
- **Is this a safe state?**

  To answer this question, we ask an intermediate question:

  - Can any of the four processes be run to completion with the resources available?
  - That is can the difference between the maximum requirement and current allocation for any process be met with the available resources?



The figure shows the state of a system consisting of four processes and three resources.

Total amount of resources

- R1 = 9
- R2 = 3
- R3 = 6

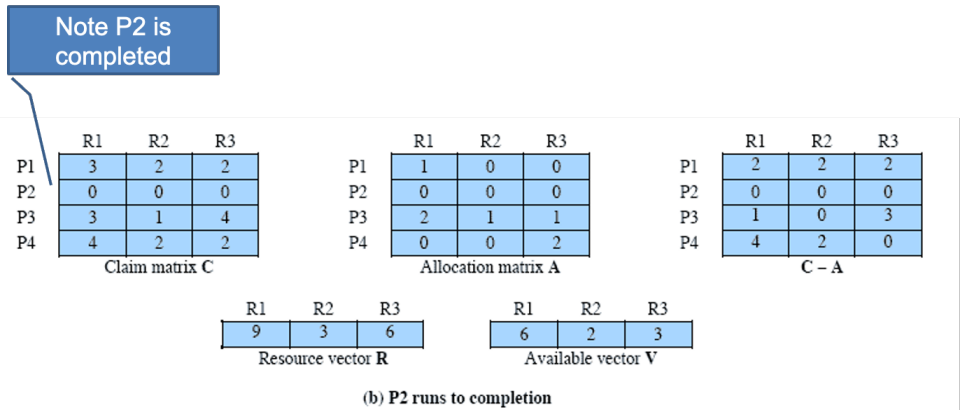
In the current state allocations have been made to the four processes, leaving available

- 1 unit of R2
- 1 unit of R3

Process i

- $C_{ij} - A_{ij} \leq V_j$, for all j
- This is not possible for P1
  - Which has only 1 unit of R1 and requires 2 more units of R1, 2 units of R2, and 2 units of R3
- If we assign one unit of R3 to process P2
  - Then P2 has its maximum required resources allocated and can run to completion and return resources to "available" pool.

After p2 runs to completion

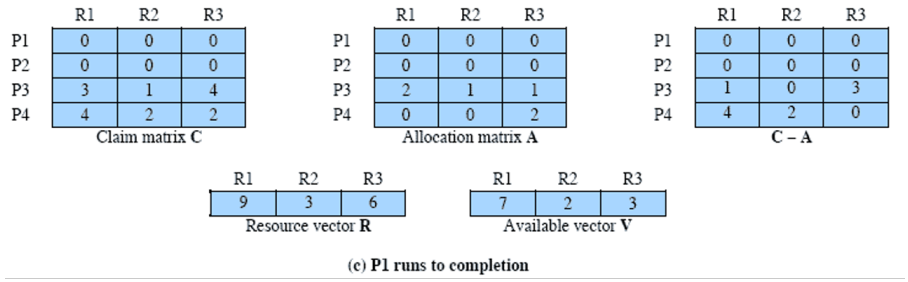


(b) P2 runs to completion

Suppose we choose P1

- allocate the required resources,
- complete P1,
- and return all of P1's resources to the available pool

After P1 completes

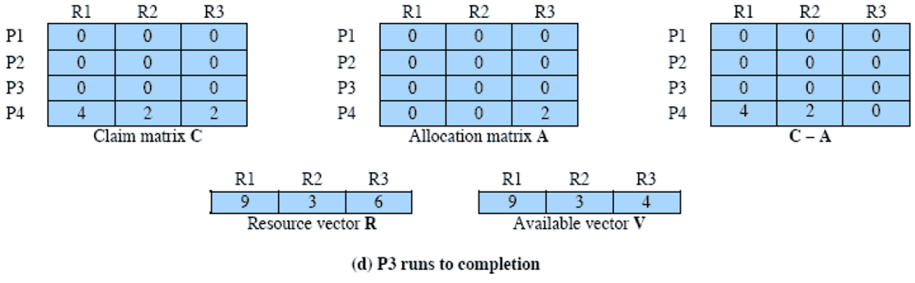


(c) P1 runs to completion

P3 Completes, resulting in the state in the following figure

Finally, we can complete P4. At this point, all of the processes have been run to completion.

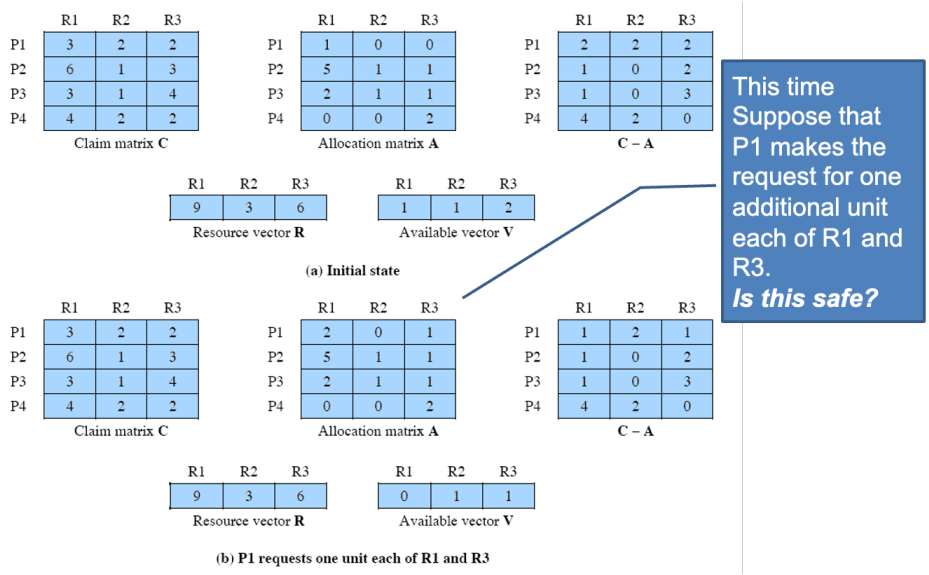
Thus, the state is a safe state.



(d) P3 runs to completion

Thus, the state defined originally is a safe state.

Determination of an Unsafe State



Suppose that P1 makes the request for one additional unit each of R1 and R3; if we assume that the request is granted,

Is this a safe state? No, because each process will need at least one additional unit of R1, and there are none available.

Thus, on the basis of deadlock avoidance, the request by P1 should be denied and P1 should be blocked.

Note: This is not a deadlock state, it merely has the potential for deadlock.

It is possible, for example, that if P1 were run from this state it would subsequently release one unit of R1 and one unit of R3 prior to needing these resources again.

- If that happened, the system would return to a safe state
- Thus, the deadlock avoidance strategy does not predict deadlock with certainty, it merely anticipates the possibility of deadlock and assures that there is never such a possibility.

Deadlock Avoidance

- When a process makes a request for a set of resources
  - assume that the request is granted
  - Update the system state accordingly
- Then determine if the result is a safe state.
  - If so, grant the request and
  - if not, block the process until it is safe to grant the request.

Deadlock Avoidance Logic

(a) global data structures

```
struct state {
  int resource[m];
  int available[m];
  int claim[n][m];
  int alloc[n][m];
}
```

(b) resource alloc algorithm

```
if (alloc [i,*] + request [*] > claim [i,*])
  < error >;                                /* total request > claim */
else if (request [*] > available [*])
  < suspend process >;
else{                                       /* simulate alloc */
  < define newstate by;
  allloc [i, *] = alloc [i, *] + request [*];
  availabel [*] = available [*] - request [*] >;
  available [*] = available [*] - request [*] >;
}
if (safe (newstate))
  < carray out allocation >;
else {
  < restore original state >;
  < suspend process >;
}
```

Above gives an abstract version of the deadlock avoidance logic. The main algorithm is shown in part(b)

With the state of the system defined by the data structure, request[*] is a vector defining the resources requested by process i.

First check is made to assure that the request does not exceed the original claim of the process.

- If the request is valid, the next step is to determine if it is possible to fulfill the request(i.e., there are sufficient resources available).
    - If it is not possible, then the process is suspended.
    - If it is possible, the final step is to determine if it is safe to fulfill the request. To do this, the resources are tentatively assigned to process i to from newstate.

(c) test for safety algorithm (banker's algorithm)

```
boolean safe (state S){
  int currentavail[m],
  process rest[<number of processes>]1
  currentavail = available;
  possible = true;
  while (possible){
    <find a process Pk in rest such that
```

```
      claim [k, *] - alloc [k, *] <= currentavail;>
    if (found){        /* simulate execution of pk */
      currentavail = currentavail + alloc [k, *];
      rest = rest - {Pk};
    }
    else possible = false;
  }
  return (rest == null);
}
```

Deadlock Avoidance Advantages

- It is not necessary to preempt and roll back processes, as in detection
- It is less restrictive than deadlock prevention

Deadlock Avoidance Restrictions

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources.

Deadlock Detection

- Deadlock prevention strategies are very conservative;
  - limit access to resources and impose restrictions on processes.
- Deadlock detection strategies do the opposite
  - Resource requests are granted whenever possible.
  - Regularly check for deadlock

发生Deadlock的需要**同时**满足如下四种条件:

1. Mutual Exclusion (互斥): 线程对于需要的资源进行互斥的访问(例如一个线程抢到锁)
2. Hold and wait (持有并等待): 线程持有了资源 (例如已将持有的锁), 同时又在等待其他资源 (例如, 需要获得锁)
3. No Preemption(非抢占式): 线程获得的资源(例如锁), 不能被抢占
4. Circular Wait (循环等待): 线程之间存在一个环路, 环路上每个线程都额外持有一个资源, 而这个资源又是下一个线程要申请的.

--> 只要一个条件没有满足就可以破解Deadlock.

Deadlock Prevention: 需要对四种条件, 进行不同的策略.

1. 打破Mutual Exclusion(互斥): Must supported by the OS.

   In general, the this condition is hard to break:

- If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS.
- Some resources, such as files, may allow multiple accesses for read for reads but only exclusive access for writes.
- Even in this case, deadlock can occur if more than one process requires write permission.

在Three easy pieces 中: 通常来说, 代码会存在临界区, 因此很难避免互斥. 但是Herlihy提出了设计各种无等待(wait-free)数据结构的思想. 想法很简单, 通过强大的硬件指令, 可以构造出不需要锁的数据结构. 详细请参阅P286

https://www.jiangguo.net/c/q3y/qy.html 活锁: 进入活锁的进程是没有阻塞的, 会继续使用CPU, 但外界看到整个进程都没有前进.

e.g.: 两个人相向过独木桥, 他们同时向一边谦让, 这样两个人都过不去, 然后又同时移到另一边, 这样两个人又过不去了. 如果不受其他因素干扰, 两个人一直同步在移动, 但外界看来两个人都没有前进, 这就是活锁.

活锁会导致CPU耗尽的, 解决办法是引入随机变量、增加重试次数等.

所以活锁也是程序设计上可能存在的问题, 导致进程都没办法运行下去了, 还耗尽CPU.

2. 打破Hold and Wait:

   Request a process request all of its required resources at one time.

```
lock(prevention);
lock(L1);
lock(L2);
...
unlock(prevention);
unlock(L1);
unlock(L2);
```

   This approach is inefficient in two ways.

   - a process may be held up for a long time waiting for all of its resources to be filled, when in fact it could have proceeded with only some of the resources.
   - resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes.

   Another problem is that a process may not know in advance all of the resources that it will require.

   There is also the practical problem created by the use of modular programming or a multithreaded structure for an application. An application would need to be aware of all resources that will be requested at all levels or in all modules to make the simultaneous request.

3. 打破 No Preemption:

   - Process must release resource and request again

     If a process holding certain resources is denied a further request, that process must release its original resources, and if necessary, request them again together with the additional resource.

   - OS may preempt a process to require it releases its resources

     If a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources.

This latter scheme would prevent deadlock only if no two processes possessed the same priority. This approach is practical only with resources whose state can be easily saved and restored later, as is the case with a processor.

`trylock()` 函数会尝试获得锁, 活着返回-1, 表示锁已经被占有, 你可以稍后重试下.

```
top:
  lock(L1);
  if (trylock(L2) == -1){
    unlock(L1);
    goto top;
  }
```

Notes: 另一个线程可以使用相同的加锁方式, 但是不同的加锁顺序(L2 然后 L1), 程序仍然不会产生死锁. 但是会引来新的问题: 活锁. 两个线程有可能一直重复这一序列, 由同时都抢锁失败. 这种情况下, 系统一直在运行这段代码(因此不是死锁), 但是又不会有进展, 因此名为活锁. 也有活锁的解决方法: 例如, 可以在循环结束的时候, 先随机等待一个时间, 然后再重复这个动作, 这样可以降低线程之间的重复互相干扰. 关于这个方案的最后一点: 使用trylock方法可能会有些困难: it skirts around the hard parts of using a trylock approach. The first problem that would likely exit again arises due to encapsulation: if one of these locks is buried in some routine that is getting called, the jump back to the beginning becomes more complex to implement. If the code had acquired some resources (other than L1) along the way, it must make sure to carefully release them as well; for example, if after acquiring L1, the code had allocated some memory, it would have to release that memory upon failure to acquire L2, before jumping back to the top to try the entire sequence again. However, in limited circumstances(e.g., the Java vector method above), this type of approach could work well. P285

4. Circular Wait
    - Define a linear ordering of resource types.

    As with hold-and-wait prevention, circular-wait prevention may be inefficient, slowing down processes and denying resources access unecessarily.