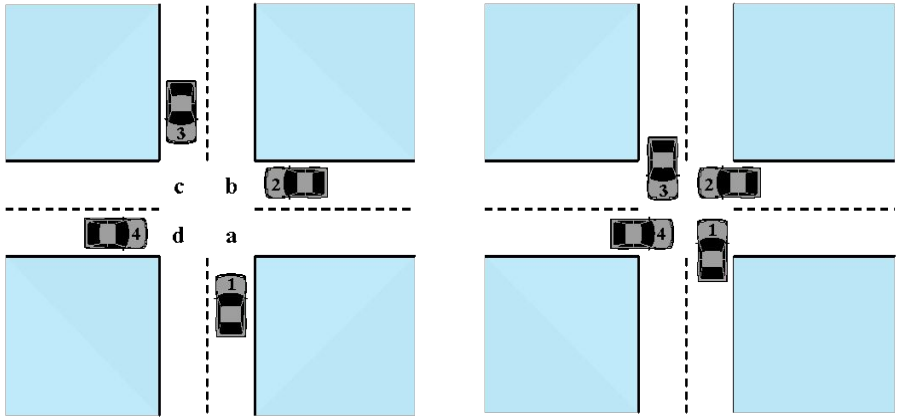
Operating Systems
CS-C3140, Lecture 8
Deadlock and Starvation

Alexandru Paler

Announcements

- The exam will not include any questions from the C programming assignments
- In order to pass the assignments 50% of the total points of six (out of eight) assignments has to be passed
  - 8 x 28 = 224 points maximum
  - 6 x 14 = 84 points needed to pass
- There will be a bonus assignment
  - number nine
  - points are added to the regular ones

Deadlock and Everyday Examples



(a) Deadlock possible

(b) Deadlock

- The *permanent* blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent because none of the events is ever triggered
- No efficient solution in the general case

Joint Progress Diagrams

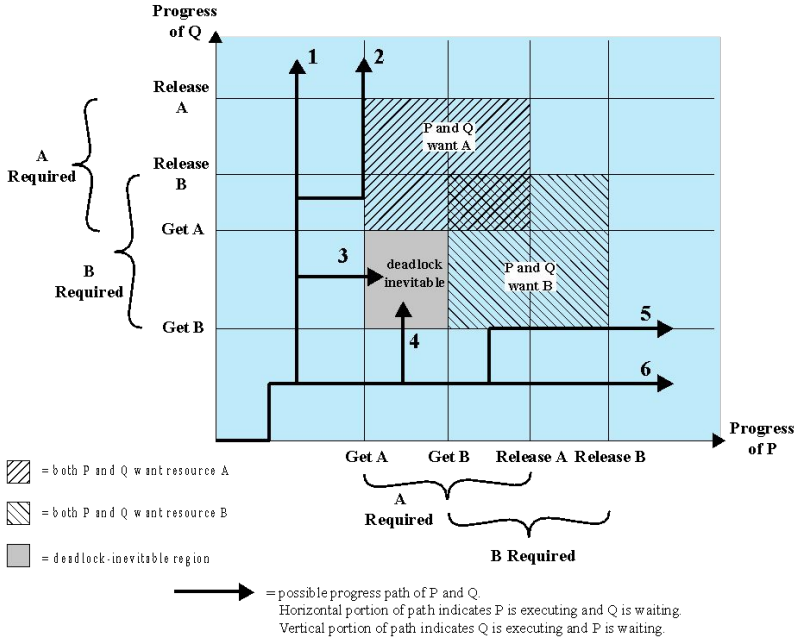


Figure 6.2 Example of Deadlock

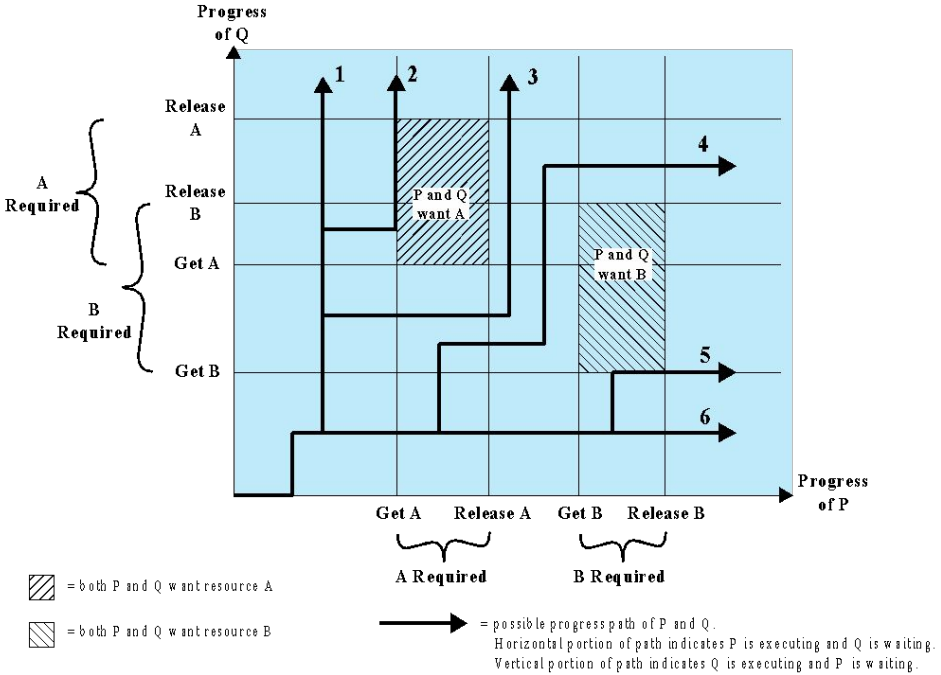


Figure 6.3 Example of No Deadlock

Resource Categories

Reusable

- Can be safely used by only one process at a time and is not depleted by that use
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Example:
  - Space is available for allocation of 200Kbytes
  - Deadlock occurs if both processes progress to their second request

P1	P2
… Request 80 Kbytes … Request60 Kbytes;	… Request 70 Kbytes … Request 80 Kbytes;

Consumable

- One that can be created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information, I/O buffers
- Example:
  - Deadlock occurs if the Receive is blocking (i.e., the receiving process is blocked until the message is received).

P1	P2
…	…
Receive (P2);	Receive (P1);
…	…
Send (P2, M1);	Send (P1, M2);

Resource Allocation Graphs

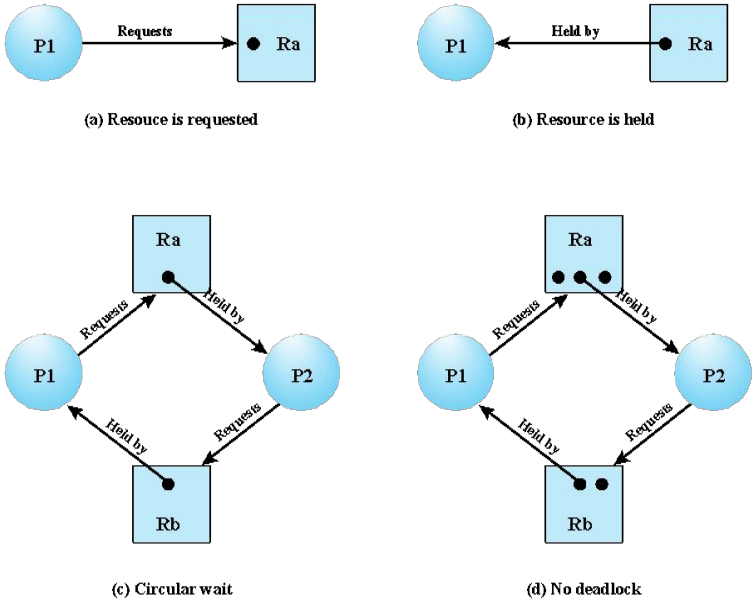


Figure 6.5 Examples of Resource Allocation Graphs

A directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node.

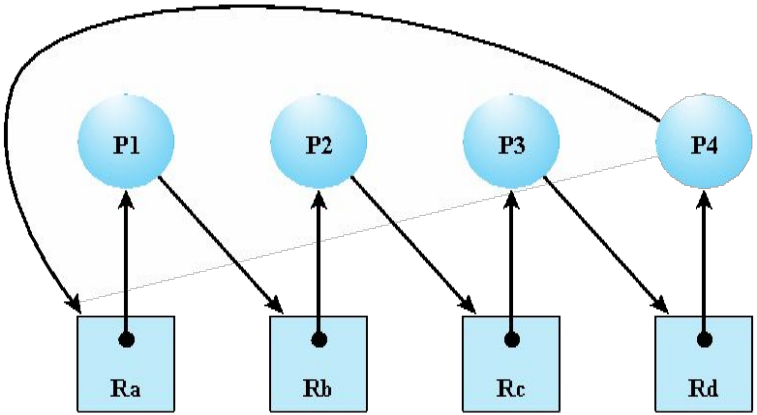
A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted.

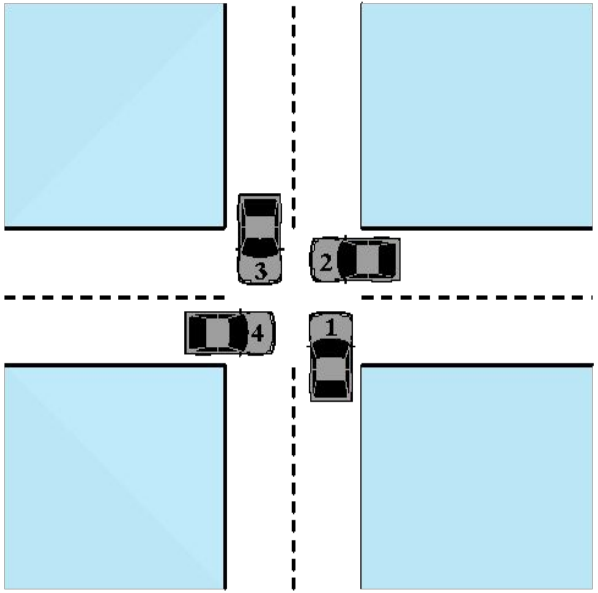
- Within a resource node, a dot is shown for each instance of that resource.
- A graph **edge directed from a resource** node dot to a process indicates a request that has been granted; that is, the **process has been assigned one unit** of that resource.

Figure 6.5c shows an example deadlock. There is only one unit each of resources Ra and Rb. Process P1 holds Rb and requests Ra, while P2 holds Ra but requests Rb.

Figure 6.5d has the same topology as Figure 6.5c , but there is no deadlock because multiple units of each resource are available.

Deadlock: Cycle in resource allocation graph





(b) Deadlock

Conditions for Deadlock

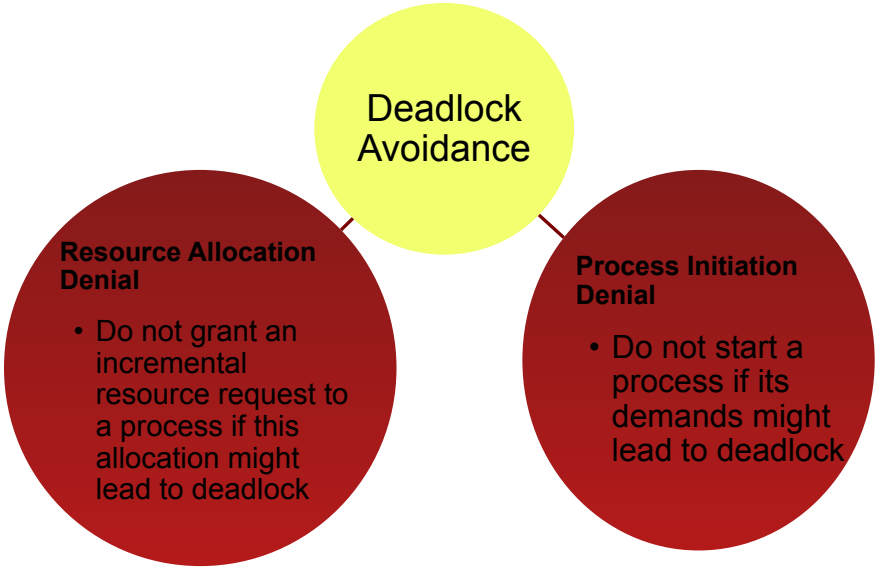
Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• Only one process may use a resource at a time • No process may access a resource until that has been allocated to another process	• A process may hold allocated resources while awaiting assignment of other resources	• No resource can be forcibly removed from a process holding it	• A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Three Approaches wrt Deadlocks

- There is no single effective strategy that can deal with all types of deadlock
- Deadlock avoidance
  - Do not grant a resource request if this allocation might lead to deadlock
- Deadlock prevention
  - Disallow one of the three necessary conditions for deadlock occurrence
  - Prevent circular wait condition from happening
- Deadlock detection
  - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover
  - A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur
  - Advantage: It leads to early detection; The algorithm is relatively simple
  - Disadvantage: Frequent checks consume considerable processor time

Deadlock Avoidance

- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- Requires knowledge of future process requests
- Advantages
  - It is not necessary to preempt and rollback processes, as in deadlock detection
  - It is less restrictive than deadlock prevention



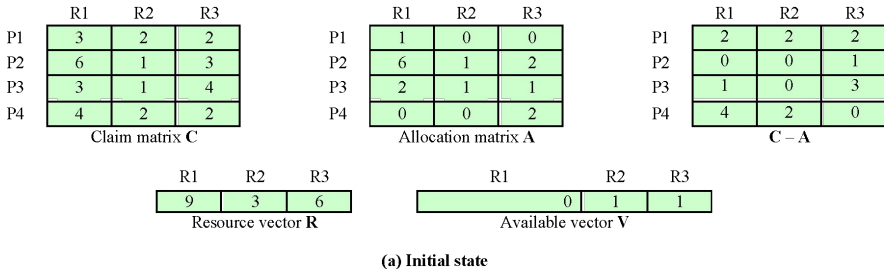
Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded
- Two main methods:
  - Indirect
    - Prevent the occurrence of one of the three necessary conditions: mutual exclusion, hold-and-wait, no pre-emption
  - Direct
    - Prevent the occurrence of a circular wait

Deadlock Condition Prevention

- Mutual exclusion
  - 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 reads but only exclusive access for writes
  - Even in this case, deadlock can occur if more than one process requires write permission
- Hold and wait
  - Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
- No Preemption
  - If a process holding certain resources is denied a further request, that process must release its original resources and request them again
  - OS may preempt the second process and require it to release its resources
- Circular Wait
  - The circular wait condition can be prevented by defining a linear ordering of resource types

Resource Allocation Denial



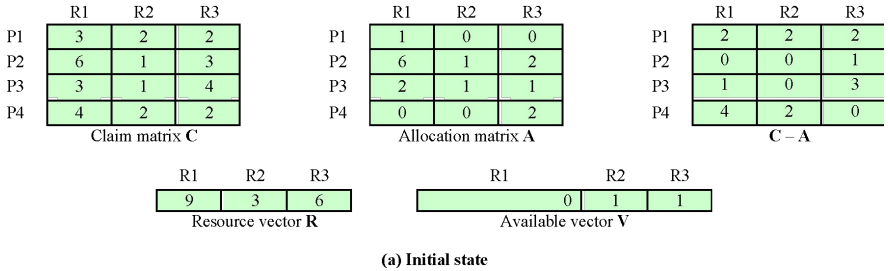
(a) Initial state

- Referred to as the banker's algorithm
- **State** of the system reflects the current allocation of resources to processes
  - The state consists of the two vectors, Resource and Available, and the two matrices, Claim and Allocation
- **Safe state** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock
- **Unsafe state** is a state that is not safe

The deadlock avoidance strategy

- does not predict deadlock with certainty
- it anticipates the possibility of deadlock
- assures that there is never such a possibility

Resource Allocation Denial: Determination of a Safe State



	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Four processes and three resources.

- The total amount of resources R1, R2, and R3 are 9, 3, and 6 units, respectively.
- In the current state allocations have been made to the four processes, leaving 1 unit of R2 and 1 unit of R3 available.

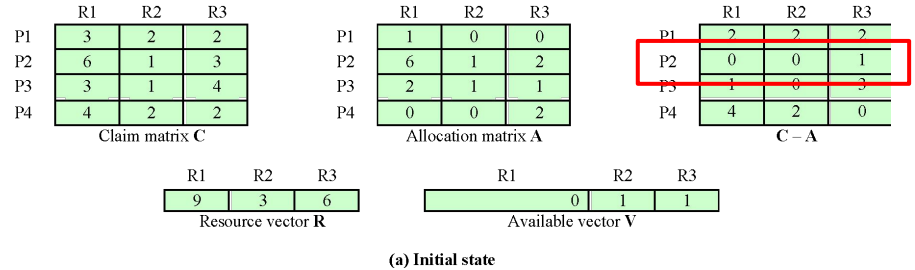
Is this a safe state?

- We ask an intermediate question
- **Any of the four processes be run to completion with the resources available**?

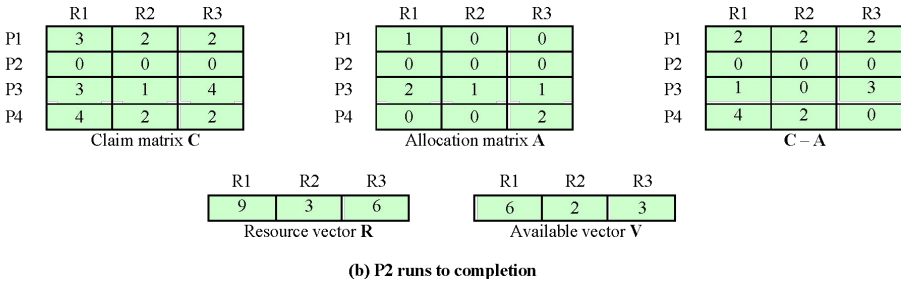
Can the **difference between the maximum requirement and current allocation for any process be met with the available resources**? In terms of the matrices and vectors introduced earlier, the condition to be met for process *i is:*

$C_{ij} - A_{ij} <= V_j$, *for all j*

Resource Allocation Denial: Determination of a Safe State



(a) Initial state



(b) P2 runs to completion

Not possible for P1

- has only 1 unit of R1
- requires 2 more units of R1, 2 units of R2, and 2 units of R3

By assigning one unit of R3 to process P2, P2 has its maximum required resources allocated and can run to completion. When P2 completes, its resources can be returned to the pool of available resources(Figure 6.7b)

Resource Allocation Denial: Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

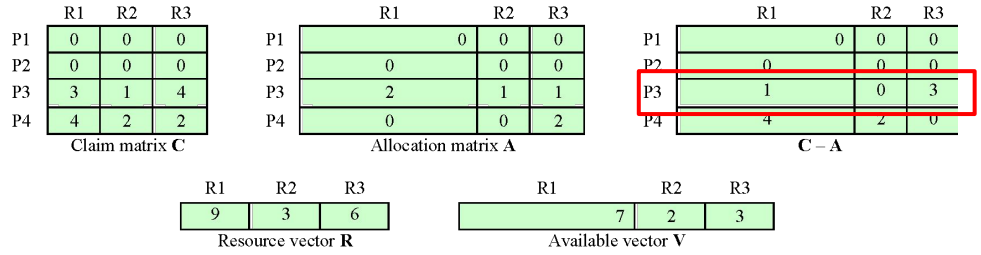
Available vector **V**

(c) P1 runs to completion

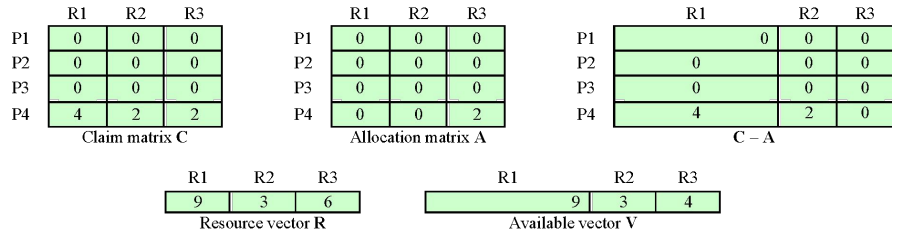
Ask again if any of the remaining processes can be completed

- Each of the remaining processes could be completed.
- Suppose we choose P1, allocate the required resources, complete P1, and return all of P1's resources to the available pool (Figure 6.7c)

Resource Allocation Denial: Determination of a Safe State



(c) P1 runs to completion



(d) P3 runs to completion

We can complete P3, resulting in the state of Figure 6.7d .

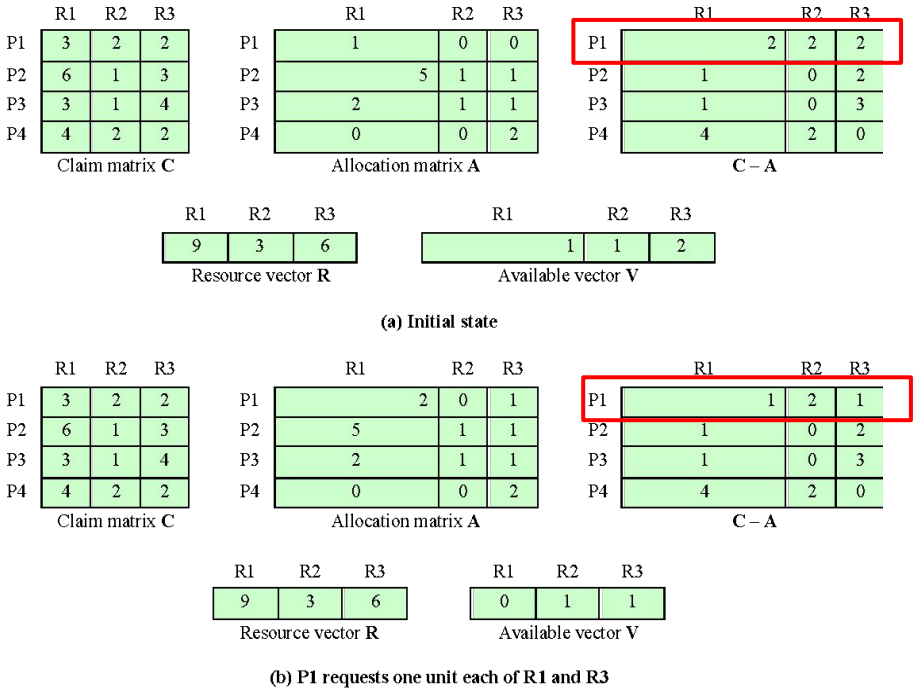
Finally, we can complete P4.

Conclusion: All of the processes have been run to completion. Thus, the state defined by Figure 6.7a is a safe state.

Deadlock avoidance strategy, which ensures that the system of processes and resources is always in a safe state:

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

Resource Allocation Denial: Determ. of an Unsafe State



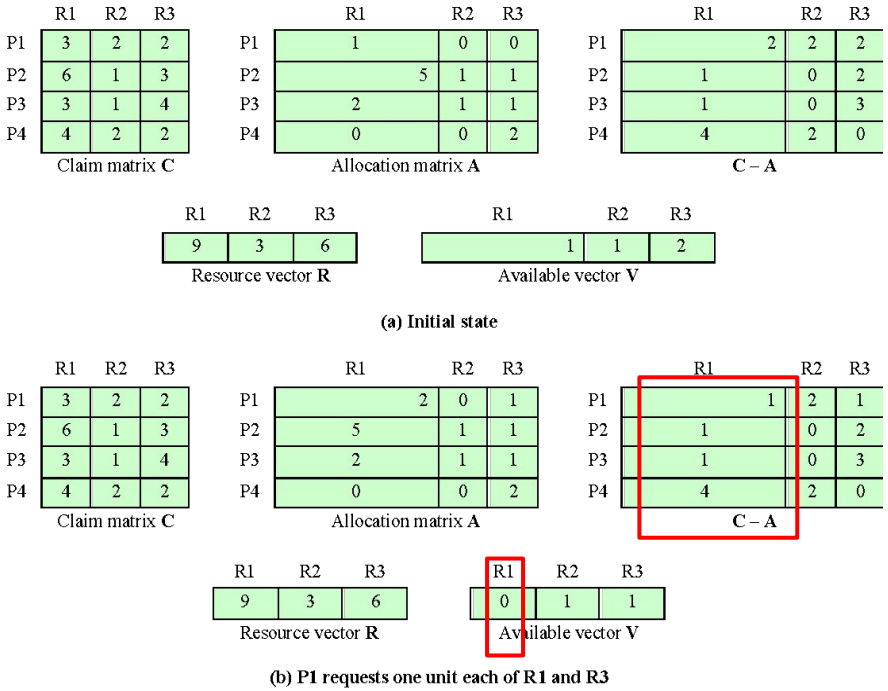
Consider the state defined in Figure 6.8a

- suppose that P1 makes the request for one additional unit each of R1 and R3;
- Is this a safe state?

The answer is 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.

It is important to point out that Figure 6.8b is not a deadlocked 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.

Resource Allocation Denial: Determ. of an Unsafe State



Consider the state defined in Figure 6.8a

- suppose that P1 makes the request for one additional unit each of R1 and R3;
- Is this a safe state?
  - Is not a deadlocked state
  - It merely has the potential for deadlock.

The answer is no, because each process will need at least one additional unit of R1, and there are none available.

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

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 Restrictions and Time Complexity

- Maximum resource requirement for each process 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

```
BOOLEAN function SAFESTATE is -- Determines if current state is safe
{ NOCHANGE : boolean;
  WORK : array[1..m] of INTEGER = AVAILABLE;
  FINISH : array[1..n] of boolean = [false, ..,false];
  I : integer;

  repeat
    NOCHANGE = TRUE;
    for I = 1 to N do
      if ((not FINISH[i]) and
         NEEDi <= WORK) then {
         WORK = WORK + ALLOCATION_i;
         FINISH[i] = true;
         NOCHANGE = false;
      }
  until NOCHANGE;
  return (FINISH == (true, .., true));
}
```

Analysis

- n Processes, m Resources
- for loop nested in a repeat loop
  - n*m
- the repeat loop can run, in worst case, n times
  - n
- Consequently n*n*m, polynomial

https://stackoverflow.com/questions/1298308/bankers-algorithm-calculated-time-complexity

Deadlock Strategies

Deadlock prevention strategies are very conservative

- Limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

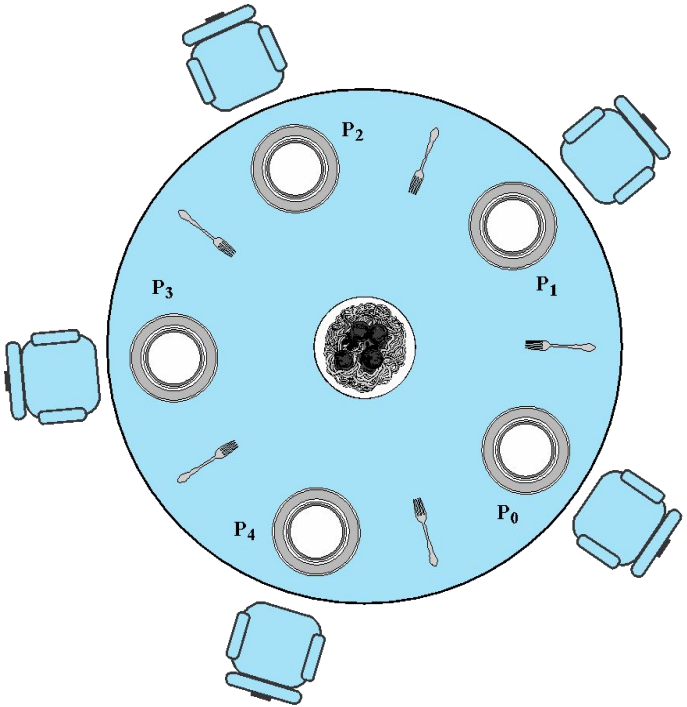
- Resource requests are granted whenever possible

Recovery Strategies

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Dining Philosophers Problem

Dining Philosophers Problem



- Five philosophers live in a house, where a table is laid for them
- The life of each philosopher consists of thinking and eating
- Each philosopher requires two forks to eat spaghetti

A philosopher wishing to eat:

- goes to a place at the table
- using the two forks on either side of the plate, takes and eats some spaghetti

Devise a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy:

- mutual exclusion (no two philosophers can use the same fork at the same time)
- avoiding deadlock and starvation

The dining philosophers problem can be seen as representative of problems dealing with the coordination of shared resources. Problem illustrates basic problems in deadlock and starvation.

Attempts to develop solutions reveal many of the difficulties in concurrent programming.

First solution to the Dining Philosophers Problem

```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Each philosopher:

- picks up first the fork on the left
- picks then the fork on the right
- finished eating and places the two forks on the table

If all of the philosophers are hungry at the same time:

- they all sit down
- they all pick up the fork on their left
- they all reach out for the other fork, which is not there
- Deadlock - All philosophers starve

To overcome the risk of deadlock:

- buy five additional forks
- teach the philosophers to eat spaghetti with one fork
- add an attendant who only allows four philosophers at a time into the dining room (next slide)

Second solution: free of deadlock and starvation

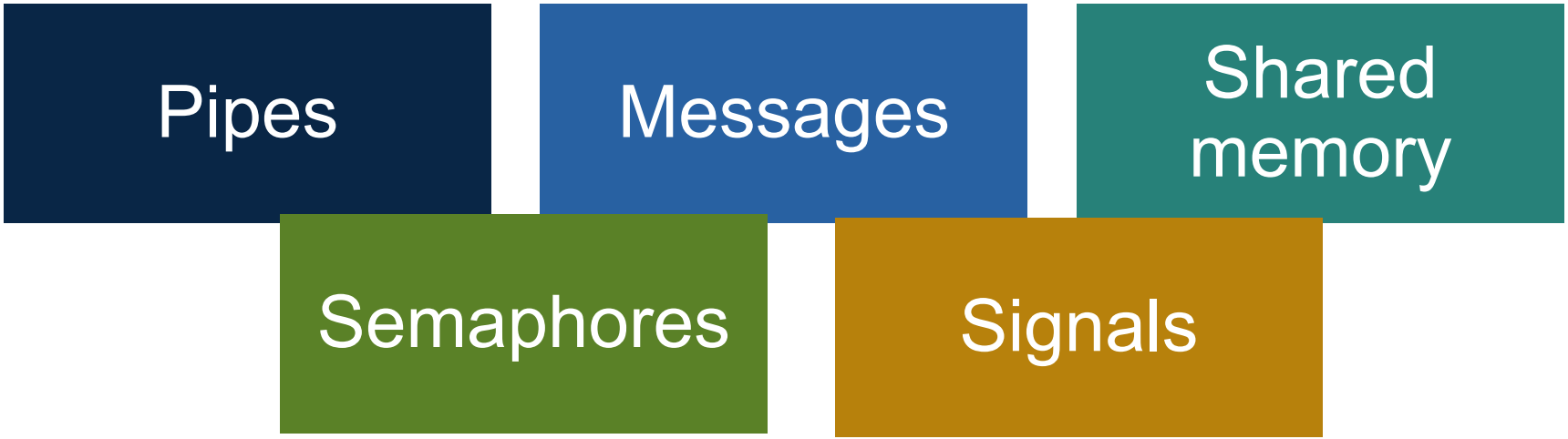
```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

https://colab.research.google.com/drive/1VY9klr7wyZoZxyBFkOB0QxquFt3x2CyC?usp=sharing

Linux Concurrency Mechanisms

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:



Pipes

- Circular buffers allowing two processes to communicate on the producer-consumer model
- First-in-first-out queue, written by one process and read by another
- Named (a special file similar to a pipe but with a name on the filesystem)
- Unnamed

Program source

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {    /* Child reads from pipe */
        close(pipefd[1]);          /* Close unused write end */

        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

https://man7.org/linux/man-pages/man2/pipe.2.html

Messages

- A block of bytes with an accompanying type
  - UNIX provides msgsnd and msgrcv system calls for processes to engage in message passing
  - The message sender specifies the type of message with each message sent, and this can be used as a selection criterion by the receiver.
- A process
  - will block when trying to send a message to a full queue.
  - will also block when trying to read from an empty queue.
  - will not block if it attempts to read a message of a certain type and fails because no message of that type is present

mq_overview(7) — Linux manual page

NAME | DESCRIPTION | NOTES | BUGS | EXAMPLES | SEE ALSO | COLOPHON

Search online pages

MQ_OVERVIEW(7) Linux Programmer's Manual MQ_OVERVIEW(7)

NAME top

mq_overview - overview of POSIX message queues

DESCRIPTION top

POSIX message queues allow processes to exchange data in the form of messages. This API is distinct from that provided by System V message queues (msgget(2), msgsnd(2), msgrcv(2), etc.), but provides similar functionality.

Message queues are created and opened using mq_open(3); this function returns a *message queue descriptor* (*mqd_t*), which is used to refer to the open message queue in later calls. Each message queue is identified by a name of the form */somename*; that is, a null-terminated string of up to **NAME_MAX** (i.e., 255) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same queue by passing the same name to mq_open(3).

Messages are transferred to and from a queue using mq_send(3) and mq_receive(3). When a process has finished using the queue, it closes it using mq_close(3), and when the queue is no longer required, it can be deleted using mq_unlink(3). Queue attributes can be retrieved and (in some cases) modified using mq_getattr(3) and mq_setattr(3). A process can request asynchronous notification of the arrival of a message on a previously empty queue using mq_notify(3).

Shared Memory

- Fastest form of interprocess communication
- Common block of virtual memory shared by multiple processes
- Permission is read-only or read-write for a process
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
  // Our memory buffer will be readable and writable:
  int protection = PROT_READ | PROT_WRITE;

  // The buffer will be shared (meaning other processes can access it), but
  // anonymous (meaning third-party processes cannot obtain an address for it),
  // so only this process and its children will be able to use it:
  int visibility = MAP_SHARED | MAP_ANONYMOUS;

  // The remaining parameters to `mmap()` are not important for this use case,
  // but the manpage for `mmap` explains their purpose.
  return mmap(NULL, size, protection, visibility, -1, 0);
}
```

```c
#include <string.h>
#include <unistd.h>

int main() {
  char parent_message[] = "hello";  // parent process will write this message
  char child_message[] = "goodbye"; // child process will then write this one

  void* shmem = create_shared_memory(128);

  memcpy(shmem, parent_message, sizeof(parent_message));

  int pid = fork();

  if (pid == 0) {
    printf("Child read: %s\n", shmem);
    memcpy(shmem, child_message, sizeof(child_message));
    printf("Child wrote: %s\n", shmem);

  } else {
    printf("Parent read: %s\n", shmem);
    sleep(1);
    printf("After 1s, parent read: %s\n", shmem);
  }
}
```

https://stackoverflow.com/questions/5656530/how-to-use-shared-memory-with-linux-in-c

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
- Similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
  - Performing some default action
  - Executing a signal-handler function
  - Ignoring the signal

Synopsis

kill [-**s** *signal*|-**p**] [--] *pid*...
kill -**l** [*signal*]

Description

The command **kill** sends the specified signal to the specified process or process group. If no signal is specified, the TERM signal is sent. The TERM signal will kill processes which do not catch this signal. For other processes, it may be necessary to use the KILL (9) signal, since this signal cannot be caught.

Most modern shells have a builtin kill function, with a usage rather similar to that of the command described here. The '-a' and '-p' options, and the possibility to specify pids by command name is a local extension.

If sig is 0, then no signal is sent, but error checking is still performed.

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

Atomic Operations and Spinlocks

- Atomic operations execute without interruption and without interference
- Simplest of the approaches to kernel synchronization
- Two types

Integer Operations

Operate on an integer variable

Typically used to implement counters

Bitmap Operations

Operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable

- Most common technique for protecting a critical section in Linux
- Can only be acquired by one thread at a time
- Any other thread will keep trying (spinning) until it can acquire the lock
- Built on an integer location in memory that is checked by each thread before it enters its critical section
- Effective in situations where the wait time for acquiring a lock is expected to be very short
- Disadvantage: Locked-out threads continue to execute in a busy-waiting mode
  - spin_lock(&lock)
  - /* critical section */
  - spin_unlock(&lock)

Memory Barriers

1. The barriers relate to machine instructions, namely loads and stores. Thus the higher-level language instruction a = b involves both a load (read) from location b and a store (write) to location a .

2. The rmb, wmb, and mb operations dictate the behavior of both the compiler and the processor.

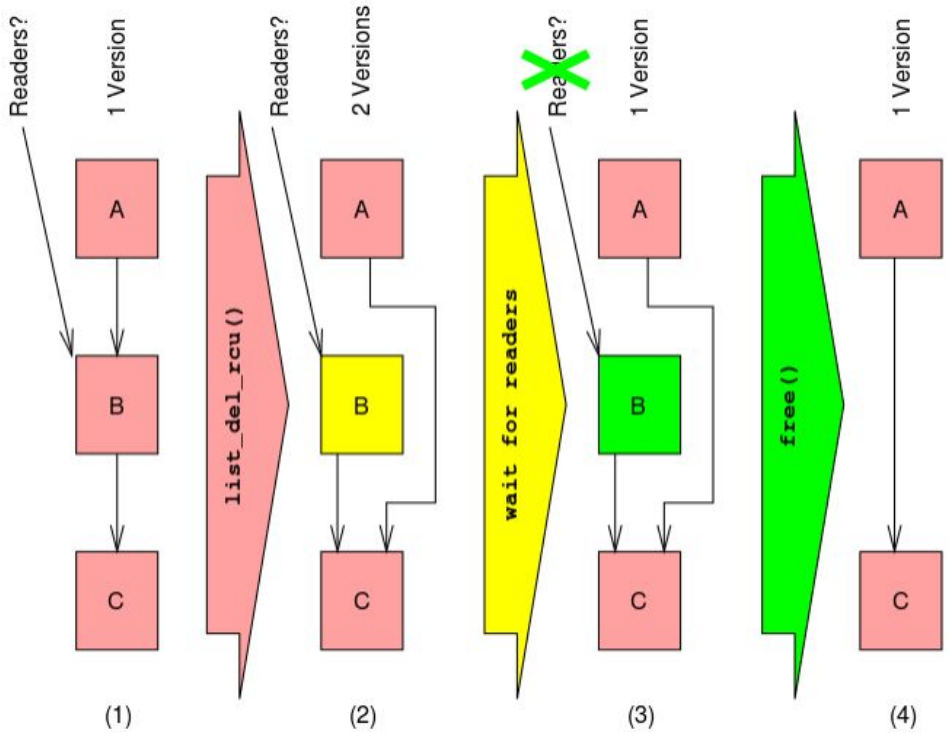
In the case of the compiler, the barrier operation dictates that the compiler not reorder instructions during the compile process.

In the case of the processor, the barrier operation dictates that any instructions pending in the pipeline before the barrier must be committed for execution before any instructions encountered after the barrier.

rmb()	Prevents loads from being reordered across the barrier
wmb()	Prevents stores from being reordered across the barrier
mb()	Prevents loads and stores from being reordered across the barrier
Barrier()	Prevents the compiler from reordering loads or stores across the barrier
smp_rmb()	On SMP, provides a rmb() and on UP provides a barrier()
smp_wmb()	On SMP, provides a wmb() and on UP provides a barrier()
smp_mb()	On SMP, provides a mb() and on UP provides a barrier()

Read-Copy-Update (RCU)

- Integrated into the Linux kernel in 2002
- The shared resources that the RCU mechanism protects must be accessed via a pointer
- The RCU mechanism provides access for multiple readers and writers to a shared resource
  - create a new structure,
  - copy the data from the old structure into the new one, and save a pointer to the old structure,
  - modify the new, copied, structure,
  - update the global pointer to refer to the new structure,
  - sleep until the operating system kernel determines that there are no readers left using the old structure, for example, in the Linux kernel, by using *synchronize_rcu()*,
  - once awakened by the kernel, deallocate the old structure.



https://en.wikipedia.org/wiki/Read-copy-update

Summary

- Principles of deadlock
  - Reusable/consumable resources
  - Resource allocation graphs
  - Conditions for deadlock
- Deadlock prevention
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Deadlock avoidance
  - Process initiation denial
  - Resource allocation denial
- Deadlock detection
  - UNIX concurrency mechanisms
    - Pipes
    - Messages
    - Shared memory
    - Semaphores
    - Signals
  - Linux kernel concurrency mechanisms
    - Atomic operations
    - Spinlocks
    - Semaphores
    - Barriers