Operating Systems 2017/2018 | Part IV – Process Synchronization

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

Slides based on the book

'Operating System Concepts, 9th Edition,

Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley'

Chapters 5 and 7

Background

- A cooperating process is one that can affect or be affected by other processes executing in the system
  - Cooperating processes can either directly share a logical address space or be allowed to share data through files or messages
- Concurrent access to shared data may result in **data inconsistency**
  - One process may be interrupted at any point in its instruction stream, partially completing its execution
  - Maintaining data consistency requires mechanisms to **ensure the orderly execution of cooperating processes**

Critical Sections and Race Conditions

- A critical section is a **piece of code** that **accesses a shared resource**
  - Code changing common variables, updating a table, writing a file, …
- When one process is executing in a critical section, no other process may be executing in the same critical section, that is, **no two processes may be executing in the same critical sections at the same time**
  - This requires that the processes be **synchronized** in some way
- A race condition occurs when several processes are allowed to access and manipulate a shared resource and the **outcome depends on the particular order in which the access takes place**, most times leading to a surprising and undesirable result

Atomic Operations

- To handle concurrency, we need to know what are the underlying **atomic operations**
  - Atomic operations are **indivisible**, i.e., they cannot be stopped in the middle of the execution and they cannot be modified by someone else in the middle of the execution
- Atomic operations are a **fundamental building block** since without them there is no way to support concurrency
  - On most machines, **memory references and assignments of words (i.e. loads and stores) are atomic operations**

Too Much Milk Problem

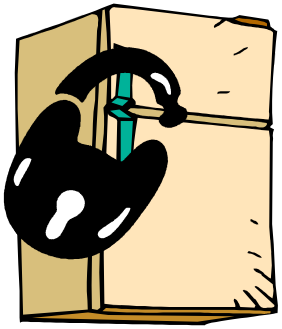
- Consider the following constraints for the problem:
  - When needed, someone buys milk
  - Never more than one person buys milk

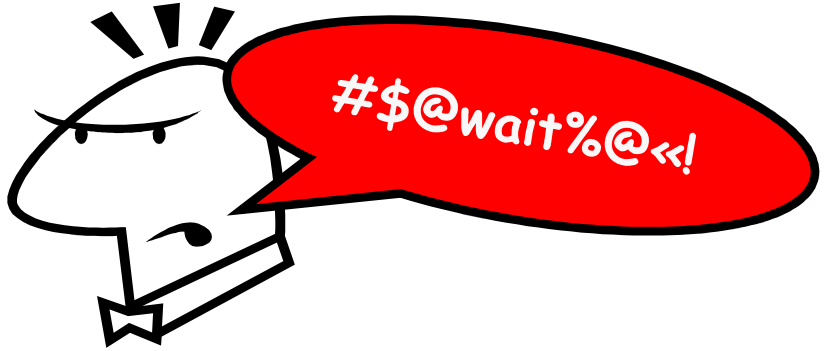
Time	Person A	Person B
15:00	Look in fridge, out of milk	
15:05	Leave for store	
15:10	Arrive to store	Look in fridge, out of milk
15:15	Buy milk	Leave for store
15:20	Arrive home, put milk away	Arrive to store
15:25		Buy milk
15:30		Arrive home, too much milk!

Lock Concept

- Problem can be fixed by using a lock around the critical region:

```
lock(fridge);
if (no milk)
  buy milk;
unlock(fridge);
```





- **Locks prevents someone from doing something**
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving and after accessing shared data
  - **Wait if locked** (somehow **synchronization involves waiting**)

- What about solutions without locks, is it possible to solve the milk problem by using only atomic load and store operations as building blocks?

Too Much Milk Problem: Solution #1

- Use a note to avoid buying too much milk
  - Leave a note before leave for store (kind of lock)
  - Remove note after arrive home (kind of unlock)
  - Don't buy milk if note exists

```
if (no milk) {
  if (no note) {
    leave note;
    buy milk;
    remove note;
  }
}
```

- **Problem:** still too much milk occasionally!

Too Much Milk Problem: Solution #1

P1	P2
`if (no milk) {`	
`  if (no note) {`	
	`if (no milk) {`
	`  if (no note) {`
`    leave note;`	
`    buy milk`	
`    ...`	
	`    leave note;`
	`    buy milk`
	`    ...`

- **Problem:** P1 can get context switched after checking for milk and for note but before leaving note

Too Much Milk Problem: Solution #2

- How about labeled notes?
  - Leave labeled note before checking

```
leave note Ni;
if (no note Nj) {
  if (no milk)
    buy milk;
}
remove note Ni;
```

- **Problem:** possible for neither process to buy milk!

Too Much Milk Problem: Solution #2

P1	P2
leave note N1;	

	leave note N2;
	if (no note N1) {
	  if (no milk)
	  buy milk;
	}

if (no note N2) {
  if (no milk)
  buy milk;
}	

...	...

- **Problem:** P1 can get context switched after leaving note but before checking P2's note

Too Much Milk Problem: Solution #3

- Another labeled note solution:

P1

```
leave note N1;
while (note N2) { // X
  do nothing;
}
if (no milk)
  buy milk;
remove note N1;
```

P2

```
leave note N2;
if (no note N1) { // Y
  if (no milk)
    buy milk;
}
remove note N2;
```

- Both programs guarantee that either it is safe to buy or the other will buy
  - At moment **X**, if no note N2 exists, it is safe for P1 to buy, otherwise wait to find out what will happen
  - At moment **Y**, if no note N1 exists, it is safe for P2 to buy, otherwise P1 is either buying or waiting for P2 to quit

Too Much Milk Problem: Solution #3

- **Problem:** solution works but it is unsatisfactory!
  - Really complex – hard to convince yourself that this code really works
  - P1's code is different from P2's code – if lots of processes, code would have to be different for each one?
  - P1's solution uses **busy waiting** – while waiting, it is consuming CPU time
- Is there a better way?

Critical Section Problem

- The goal of the **critical section problem** is to design a protocol that processes can use to cooperate:
  - Processes must ask permission to enter the critical section, the **entry section**
  - The critical section may then be followed by an **exit section** and/or by a **remainder section** (non-critical section)

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Critical Section Problem

- A solution to the critical section problem must satisfy 3 requirements:
  - **Mutual exclusion** – if a process is executing in a critical section, then no other processes can be executing in the same critical section
  - **Progress** – if no process is executing in a critical section, then only those processes that are not executing in a remainder section can participate in deciding which will enter the critical section next, and this decision cannot be postponed indefinitely
  - **Bounded waiting** – a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Is This Code a Solution?

```
// shared variables
boolean lock = false;
...
// two processes P0 and P1
do {
   while (lock);
   lock = true;
    // critical section
   lock = false;
    // non-critical section
} while (true);

// not a solution: mutual exclusion not preserved
```

Is This Code a Solution?

```
// shared variables
int turn = 0;
...
// two processes P0 and P1
// Pi is the current process
do {
  while (turn != i);
   // critical section
  turn = 1 - i;
   // non-critical section
} while (true);

// not a solution: progress requirement not satisfied
```

Peterson's Solution

- **Classic software-based solution** to the critical section problem:
  - Provides a good algorithmic description of solving the problem
  - Illustrates the complexities involved in designing software that addresses the requirements of mutual exclusion, progress and bounded waiting
  - Is a **two process solution**, but has a generalization for N processes
- Provable that:
  - Mutual exclusion is preserved
  - Progress requirement is satisfied
  - Bounded waiting requirement is met
- Uses two shared variables:
  - Variable **turn** indicates whose turn it is to enter the critical section
  - Array **flag[2]** indicates if a process is ready to enter the critical section

Peterson's Solution

```
// shared variables
int turn; boolean flag[2] = {false, false};
...
// two processes P0 and P1
// Pi is the current process and Pj is the other
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn = = j);  // busy waiting
    // critical section
   flag[i] = false;
    // non-critical section
} while (true);
```

Synchronization Hardware

- In uniprocessor systems, the critical section problem could be solved if we **could disable interrupts** while modifying shared variables:
  - Running code would **execute without preemption**

```
disable interrupts;
// critical section
enable interrupts;
```

- Problems with the use of interrupts:
  - Cannot allow users to use it explicitly (user might forgot to enable interrupts!)
  - Not scalable on multiprocessor systems (disabling interrupts on all processors requires messages, which would be too inefficient)

Synchronization Hardware

- Modern computer systems provide **special hardware instructions** that can be used effectively and efficiently to solve the critical section problem based on the idea of protecting critical regions via **locking**
  - **test_and_set()** – atomically test memory word and set value
  - **compare_and_swap()** – atomically swap contents of two memory words
- Unlike disabling interrupts, special hardware instructions **can be used on both uniprocessors and multiprocessors**
  - On uniprocessors not too hard
  - On multiprocessors requires help from the cache coherence protocol

Hardware Atomic Instructions

```
boolean test_and_set(boolean *target) {
  boolean temp = *target;
  *target = true;
  return temp;
}

int compare_and_swap(int *target, int expected, int new_val) {
  int temp = *target;
  if (*target = = expected)
    *target = new_val;
  return temp;
}
```

Mutual Exclusion with test_and_set()

```
// shared variable
boolean lock = false;
...
// mutual exclusion
do {
   while (test_and_set(&lock));
   // critical section
   lock = false;
   // non-critical section
} while (true);

// still, bounded waiting requirement not satisfied
```

Mutual Exclusion with compare_and_swap()

```
// shared variable
int lock = 0;
...
// mutual exclusion
do {
  while (compare_and_swap(&lock, 0, 1));
   // critical section
  lock = 0;
   // non-critical section
} while (true);

// still, bounded waiting requirement not satisfied
```

Bounded Waiting with test_and_set()

```
// shared variables
boolean lock = false, waiting[N] = {false, ..., false};
..
do {
   waiting[i] = true;  key = true;
   while (waiting[i] && key) key = test_and_set(&lock);
   waiting[i] = false;
   // critical section
   j = (i + 1) % n;
   while ((j != i) && waiting[j] = = false) j = (j + 1) % n;
   if (j = = i) lock = false else waiting[j] = false;
   // non-critical section
} while (true);
```

Mutex Locks

- Previous solutions are generally hidden from programmers
  - OS designers build software tools to solve the critical section problem
  - Simplest tool is the **mutex lock** (short for mutual exclusion)
- Mutexs can protect critical regions and prevent race conditions
  - Boolean variable indicates if the lock is available or not
  - A **init_lock()** function initializes the lock, a **acquire_lock()** function acquires the lock and a **release_lock()** function releases the lock
  - Calls to acquire_lock() and release_lock() must be atomic and, thus, they are often implemented via hardware atomic instructions

Mutex Locks

```
init_lock(mutex);
...
do {
   acquire_lock(mutex);  // busy waiting
    // critical section
   release_lock(mutex);
    // non-critical section
} while (true);

#define init_lock(M)    { M = false; }
#define acquire_lock(M) { while (test_and_set(&M)); }
#define release_lock(M) { M = false; }
```

Mutex Locks

```
init_lock(mutex);
...
do {
   acquire_lock(mutex);  // busy waiting
    // critical section
   release_lock(mutex);
    // non-critical section
} while (true);

#define init_lock(M)    { M = 0; }
#define acquire_lock(M) { while (compare_and_swap(&M, 0, 1)); }
#define release_lock(M) { M = 0; }
```

Busy Waiting

- When the mutex lock solution requires busy waiting, it is called **spinlock** because it **spins while waiting for the lock to become available**
  - Busy waiting wastes CPU time that some other process might be using
  - Usually, spinlocks do not satisfy the bounded waiting requirement
  - Case of previous approaches using test_and_set() and compare_and_swap()
- Can be **advantageous if locks are to be held for short periods of time**
  - Often employed on multiprocessor systems where one process performs its critical section on one processor, while the others spin on another processors
  - No context switch is required when waiting on a spinlock
- Can be a **problem for multiprogramming systems**
  - If the process holding the lock is waiting to run, no other process can access the lock and thus it can be useless to give CPU time to such processes

Handling Busy Waiting Time

- Can we build mutex locks without busy waiting?
  - Sorry, no!
- But, can we minimize busy waiting time?
  - Yes, by only busy waiting to atomically check lock value!
- Minimizing busy waiting time can be implemented with an **associated queue of waiting processes** and a **suspend() operation that voluntary suspends the execution of the current process**
- Two different approaches:
  - For uniprocessors via **disabling interrupts**
  - For multiprocessors via **disabling interrupts plus atomic instructions**

Uniprocessor Lock Implementation

```
// mutex data structure
typedef struct {
  int value;        // mutex's value (FREE or BUSY)
  PCB *queue;       // associated queue of waiting processes
} mutex;

init_lock(mutex M) { M.value = FREE;
                     M.queue = EMPTY; }
```

Uniprocessor Lock Implementation

```
acquire_lock(mutex M) {
  disable interrupts;
  if (M.value = = BUSY) {
    // avoid busy waiting
    add_to_queue(current PCB, M.queue);
    suspend();
    // kernel reenables interrupts just before restarting here
  } else {
    M.value = BUSY;
    enable interrupts;
  }
}
```

Uniprocessor Lock Implementation

```
release_lock(mutex M) {
  disable interrupts;
  if (M.queue != EMPTY) {
    // leave lock BUSY and wakeup one waiting process
    PCB = remove_from_queue(M.queue);
    add_to_queue(PCB, ready queue);
  } else {
    M.value = FREE;
  }
  enable interrupts;
}
```

Multiprocessor Lock Implementation

```
// mutex data structure
typedef struct {
  boolean guard;  // to guarantee atomicity
  int value;      // mutex's value (FREE or BUSY)
  PCB *queue;     // associated queue of waiting processes
} mutex;

init_lock(mutex M) { M.guard = false;
                     M.value = FREE;
                     M.queue = EMPTY; }
```

Multiprocessor Lock Implementation

```
acquire_lock(mutex M) {
  disable interrupts;
  while (test_and_set(&M.guard));  // short busy waiting time
  if (M.value = = BUSY) {
    add_to_queue(current PCB, M.queue);
    M.guard = false;
    suspend();
    // kernel reenables interrupts just before restarting here
  } else {
    M.value = BUSY;  M.guard = false;
    enable interrupts;
  }
}
```

Multiprocessor Lock Implementation

```
release_lock(mutex M) {
  disable interrupts;
  while (test_and_set(&M.guard));  // short busy waiting time
  if (M.queue != EMPTY) {
    // leave lock BUSY and wakeup one waiting process
    PCB = remove_from_queue(M.queue);
    add_to_queue(PCB, ready queue);
  } else {
    M.value = FREE;
  }
  M.guard = false;
  enable interrupts;
}
```

Semaphores

- A semaphore is a synchronization tool used to **control access** to a given **resource consisting of a finite number of instances**
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Semaphores are like integers, except:
  - **Cannot read or write value**, except to set it initially
  - **No negative values** (when a semaphore reaches 0 all instances are being used, meaning that, all processes that wish to use the resource will block until the semaphore becomes greater than 0)
- Two types of semaphores exist:
  - **Counting semaphore** which can range over an unrestricted domain
  - **Binary semaphore** which can range only between 0 and 1 (like mutex locks)

Semaphores Operations

- Semaphores are accessed through two standard **atomic operations**:
  - **wait()** operation waits the semaphore to become positive and then decrements it by one
  - **signal()** operation increments the semaphore by one

```
wait(semaphore S) {
  while (S = = 0);  // busy waiting
  S--;
}

signal(semaphore S) {
  S++;
}
```

Semaphores Implementation

- Implementation must guarantee that **no two wait() and/or signal() operations are executed on the same semaphore at the same time**
  - Simultaneous wait() operations cannot decrement value below zero
  - Cannot miss an increment from signal() if wait() operation happens simultaneously
- Again, two different approaches to **minimize busy waiting time**:
  - For uniprocessors via **disabling interrupts**
  - For multiprocessors via **disabling interrupts plus atomic instructions**

Uniprocessor Semaphore Implementation

```
// semaphore data structure
typedef struct {
  int value;        // semaphore's value
  PCB *queue;       // associated queue of waiting processes
} semaphore;

init_semaphore(semaphore S) { S.value = 0;
                              S.queue = EMPTY; }
```

Uniprocessor Semaphore Implementation

```
wait(semaphore S) {
  disable interrupts;
  if (S.value = = 0) {
    // avoid busy waiting
    add_to_queue(current PCB, S.queue);
    suspend();
    // kernel reenables interrupts just before restarting here
  } else {
    S.value--;
    enable interrupts;
  }
}
```

Uniprocessor Semaphore Implementation

```
signal(semaphore S) {
  disable interrupts;
  if (S.queue != EMPTY) {
    // keep semaphore's value and wakeup one waiting process
    PCB = remove_from_queue(S.queue);
    add_to_queue(PCB, ready queue);
  } else {
    S.value++;
  }
  enable interrupts;
}
```

Multiprocessor Semaphore Implementation

```
// semaphore data structure
typedef struct {
  boolean guard;  // to guarantee atomicity
  int value;      // semaphore's value
  PCB *queue;     // associated queue of waiting processes
} semaphore;

init_semaphore(semaphore S) { S.guard = false;
                              S.value = 0;
                              S.queue = EMPTY; }
```

Multiprocessor Semaphore Implementation

```
wait(semaphore S) {
  disable interrupts;
  while (test_and_set(&S.guard));  // short busy waiting time
  if (S.value = = 0) {
    add_to_queue(current PCB, S.queue);
    S.guard = false;
    suspend();
    // kernel reenables interrupts just before restarting here
  } else {
    S.value--;  S.guard = false;
    enable interrupts;
  }
}
```

Multiprocessor Semaphore Implementation

```
signal(semaphore S) {
  disable interrupts;
  while (test_and_set(&S.guard));  // short busy waiting time
  if (S.queue != EMPTY) {
    // keep semaphore's value and wakeup one waiting process
    PCB = remove_from_queue(S.queue);
    add_to_queue(PCB, ready queue);
  } else {
    S.value++;
  }
  S.guard = false;
  enable interrupts;
}
```

Starvation and Deadlock

- Starvation (or indefinite blocking) is a situation in which a process **waits indefinitely for an event that might never occur**
- Deadlock is a situation in which a set of processes is **waiting indefinitely for an event that will never occur** because that event can be caused only by one of the waiting processes in the set
- **Deadlock $\Rightarrow$ starvation** but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock cannot end without external intervention

Deadlock

- Deadlocks occur when **accessing multiple resources**
  - Cannot solve deadlock for each resource independently
- Deadlocks are **not deterministic** and **won't always happen** for the same piece of code
  - Has to be exactly the right timing (or wrong timing?)

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Possible deadlock

P0: wait(S);
P1: wait(Q);
P0: wait(Q);
P1: wait(S);
...

Classical Problems of Synchronization

- **Bounded buffer problem**
  - Commonly used to illustrate the power of synchronization primitives
- **Readers and writers problem**
  - Commonly used to illustrate the problem of sharing data
- **Dining philosophers problem**
  - Commonly used to illustrate the class of concurrency control problems

Bounded Buffer Problem

- Illustrates the power of synchronization primitives:
  - **Producer** – puts things into a shared buffer
  - **Consumer** – takes them out
- Correctness constraints:
  - **Mutual exclusion constraint**: only one process can manipulate the buffer queue at a time
  - **Scheduling constraint**: if buffer is full, producer must wait for consumer to make room in buffer
  - **Scheduling constraint**: if buffer is empty, consumer must wait for producer to fill buffer

Bounded Buffer Problem: Solution

- Shared data structures:
  - Variable **n** stores the number of buffers (each buffer can hold one item)
  - Semaphore **mutex** provides mutual exclusion for accesses to the buffer pool
  - Semaphore **empty** counts the number of empty buffers
  - Semaphore **full** counts the number of full buffers

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

- Key idea:
  - Use a separate semaphore for each constraint
  - Symmetry between the producer and the consumer, we can interpret the solution as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer

Bounded Buffer Problem: Producer

```
do {
    . . .
   /* produce an item in next_produced */
    . . .
   wait(empty);
   wait(mutex);
    . . .
   /* add next_produced to the buffer */
    . . .
   signal(mutex);
   signal(full);
} while (true);
```

Bounded Buffer Problem: Consumer

```
do {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
} while (true);
```

Bounded Buffer Problem: Discussion

- Is the order of the wait() calls important?
  - Yes, otherwise can cause deadlocks!
- Is the order of the signal() calls important?
  - No, except that it might affect efficiency!
- What if we have 2 producers or 2 consumers, do we need to change anything?
  - No, it works OK!

Readers and Writers Problem

- Illustrates the problem of sharing data:
  - **Readers** – only read the shared data
  - **Writers** – can both read and write the shared data
- Is a **single lock** sufficient to synchronize the access to the shared data?
  - For writers is OK since writers must have exclusive access to shared data
  - For readers is not since we may want **multiple readers at the same time**
- Useful in applications:
  - Where it is easy to identify which processes only read shared data and which processes only write shared data
  - Where exists more readers than writers and the increased concurrency of having multiple readers compensates the overhead involved in implementing the reader–writer solution

Readers and Writers Problem: Variations

- **Variation I** – no reader is kept waiting unless a writer has already obtained permission to use the shared data
  - In other words, no reader should wait for other readers to finish simply because a writer is waiting
  - May result in **starvation of the writers**
- **Variation II** – once a writer is ready, it executes as soon as possible
  - In other words, if a writer is waiting to access the shared data, no new reader may start reading
  - May result in **starvation of the readers**
- The solution that follows implements variation I

Readers and Writers Problem: Solution

- Shared data structures:
  - Semaphore **rw_mutex** ensures mutual exclusion for writers (also used by the first/last reader that enters/exits the critical section)
  - Semaphore **mutex** ensures mutual exclusion when updating variable read_count
  - Variable **read_count** keeps track of how many processes are currently reading the shared data

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

- Key idea:
  - If a writer is in the critical section and N readers are waiting, then only one reader is queued on rw_mutex, the other N–1 readers are queued on mutex
  - When a writer signals rw_mutex, we may resume the execution of either the waiting readers or a waiting writer (the selection is made by the scheduler)

Readers and Writers Problem: Writer

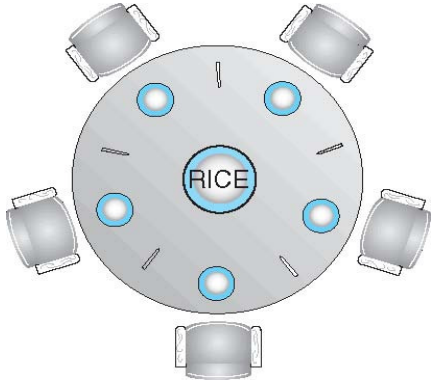
```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Readers and Writers Problem: Reader

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Dining Philosophers Problem

- Illustrates the class of concurrency control problems:
  - Simple representation of the need to allocate several resources among several processes in a **deadlock-free** and **starvation-free** manner



- A set of N philosophers share a table laid with N single chopsticks and with a bowl of rice in its center
  - When a philosopher thinks, he does not interact with their colleagues
  - From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest (one chopstick at a time)
  - When a philosopher has both chopsticks, he eats from the bowl without releasing the chopsticks
  - When he finished eating, he puts down both chopsticks and starts thinking again

Dining Philosophers Problem: Solution

- Shared data structures:
  - Bowl of rice (data)
  - Semaphore **chopstick[N]** representing the access to each chopstick

```
semaphore chopstick[5];
```

- Key idea:
  - A philosopher tries to grab his chopsticks by executing wait() operations on the appropriate semaphores
  - A philosopher releases his chopsticks by executing signal() operations on the appropriate semaphores

Dining Philosophers Problem

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Dining Philosophers Problem: Discussion

- What happens if all philosophers become hungry at the same time?
  - If all philosophers start by grabbing their left chopstick then, when they try to grab their right chopstick, they will be delayed forever, leading to a deadlock
- Possible remedies to the deadlock problem:
  - Allow at most four philosophers to be sitting simultaneously at the table
  - Use an asymmetric solution, e.g., an odd-numbered philosopher picks up first the left chopstick and then the right chopstick, whereas an even numbered philosopher picks up first the right chopstick and then the left chopstick
- A satisfactory solution must also guard against the possibility of starvation
  - Note that a deadlock-free solution does not necessarily eliminate starvation

Deadlock Characterization

- Deadlocks can arise only if the next 4 conditions hold simultaneously:
  - **Mutual exclusion** – only one process at a time can use a resource (a requesting process must be delayed until the resource has been released)
  - **Hold and wait** – a process holding (at least) one resource is waiting to acquire additional resources held by other processes
  - **No preemption** – a resource cannot be preempted, i.e., a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - **Circular wait** – there exists a set of waiting processes $\{P_1, P_2, \ldots, P_N\}$ such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3, …, P_{N-1} is waiting for a resource held by P_N, and P_N is waiting for a resource held by P_1

Resource Allocation Graph

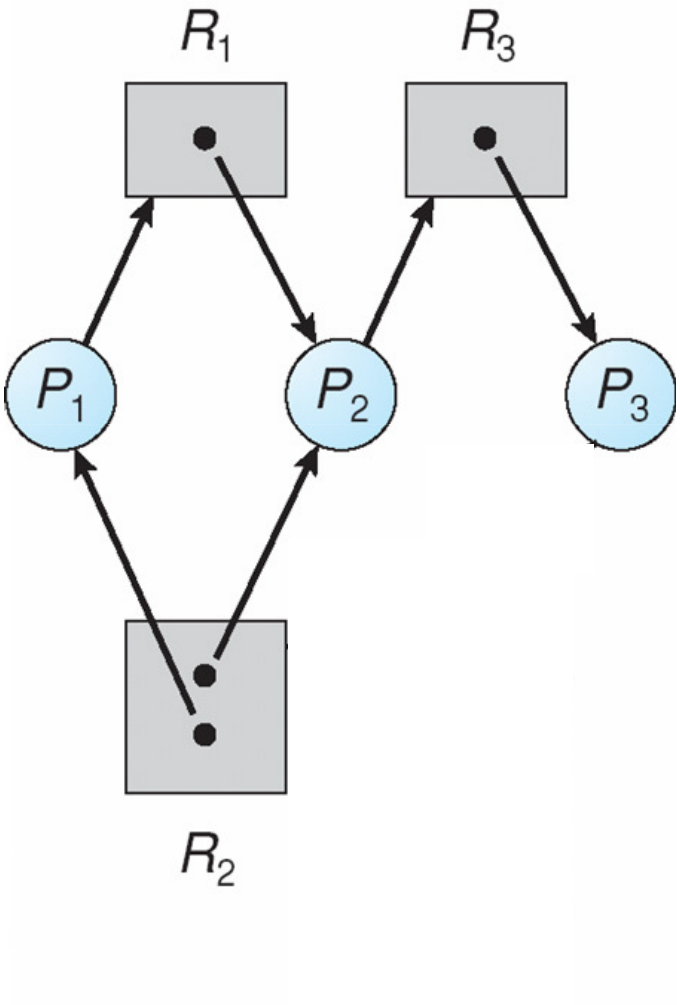
- Deadlocks can be described more precisely in terms of a **directed graph**, called a **resource allocation graph**, that consists of a **set of vertices V** and a **set of edges E**

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_N\}$, the **set of all processes** in the system
  - $R = \{R_1, R_2, \ldots, R_M\}$, the **set of all resource types** in the system

- E is partitioned into two types:
  - **Er (request edges)**, the set of all directed edges $P_i \rightarrow R_j$ meaning that process P_i has requested an instance of resource type R_j and is currently waiting for that resource
  - **Ea (assignment edges)**, the set of all directed edges $R_j \rightarrow P_i$ meaning that an instance of resource type R_j has been allocated to process P_i

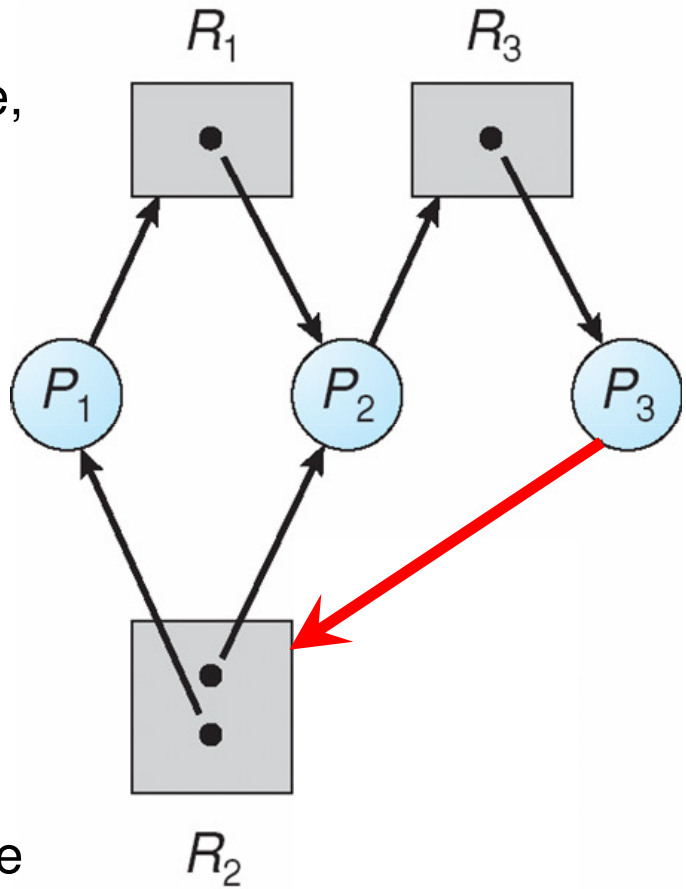
Resource Allocation Graph: Example

- The sets P, R, Er and Ea:
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3\}$
  - $Er = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3\}$
  - $Ea = \{R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3\}$
- Process states:
  - P_1 is holding one instance of R_2 and is waiting for an instance of R_1
  - P_2 is holding one instance of R_1 and one instance of R_2 and is waiting for an instance of R_3
  - P_3 is holding an instance of R_3



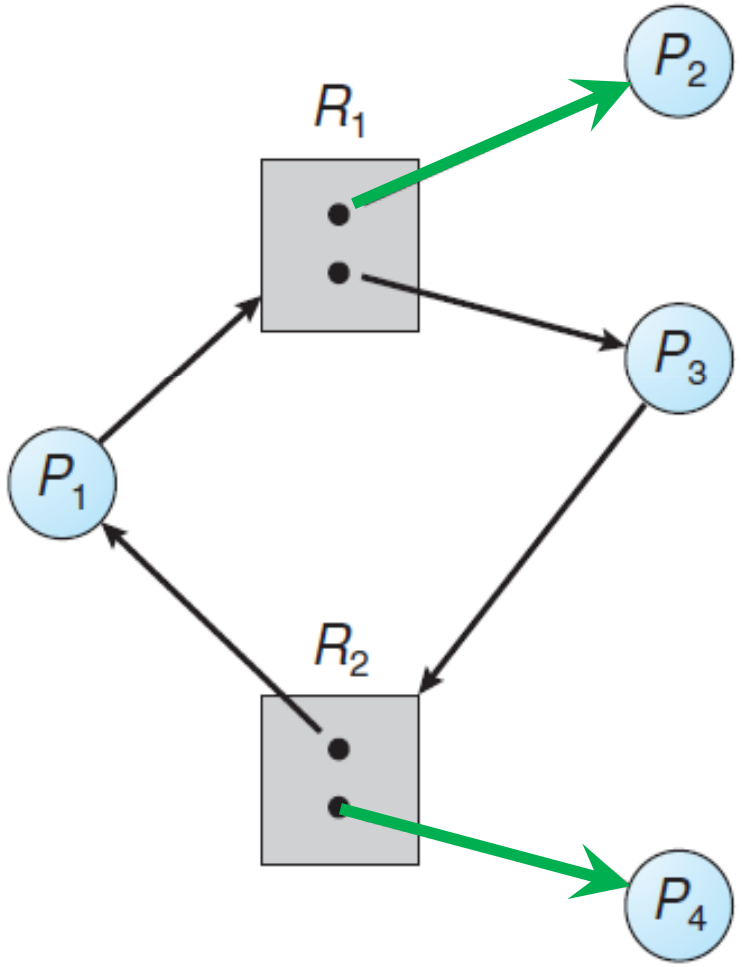
Resource Allocation Graph: Deadlock?

- P_3 requests an instance of R_2:
  - Since no resource instance is currently available, we add a request edge $P_3 \rightarrow R_2$ to the graph
- Two minimal cycles now exist in the system:
  - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
  - $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- Processes P_1, P_2 and P_3 are **deadlocked**
  - P_1 is waiting for resource R_1, which is held by P_2
  - P_2 is waiting for resource R_3, which is held by P_3
  - P_3 is waiting for either P_1 or P_2 to release resource R_2



Resource Allocation Graph: Deadlock?

- Consider another resource allocation graph with a cycle:
  - $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- Despite the cycle, there is **no deadlock**
  - P_2 may release its instance of resource type R_1 and that resource can then be allocated to P_1, breaking the cycle
  - Alternatively, P_4 may also release its instance of resource type R_2 and that resource can then be allocated to P_3, breaking the cycle



Resource Allocation Graph: Summary

- If the graph contains **no cycles ⇒ no deadlock**
- If the graph contains a **cycle ⇒ deadlock may exist**
  - If the cycle involves only resource types which have exactly a single instance, then a deadlock has occurred
  - If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred
- A cycle is thus a **necessary but not a sufficient condition** for the existence of a deadlock

Handling Deadlocks

- To ensure that deadlocks never occur, the system can use either a protocol to prevent or avoid deadlocks:
  - **Deadlock prevention** methods ensure that at least one of the four necessary conditions cannot hold by constraining how requests for resources can be made
  - **Deadlock avoidance** methods require that the operating system be given additional information in advance, concerning which resources a process will request and use during its lifetime, in order to decide whether a resource request can be satisfied or must be delayed
- If a system does not employ either a deadlock prevention or avoidance method, then a deadlock situation may arise:
  - **Deadlock detection** methods examine the state of the system to determine whether a deadlock has occurred and provide algorithms to recover from deadlocks

Handling Deadlocks

- In the absence of methods to prevent, avoid or recover from deadlocks, we may arrive at situations in which the system is in a deadlock state and has no way of recognizing what has happened
  - **Undetected deadlocks** might cause the **system's performance to deteriorate**, since resources are being held by processes that cannot run and because more and more processes, as they ask for the same resources, will enter a deadlock state (eventually, the **system will stop working** and will need to be restarted manually)

- Expense is one important consideration since ignoring the possibility of deadlocks is cheaper than the other approaches
  - If deadlocks occur infrequently, the extra expense of the other methods may not seem worthwhile
  - In addition, methods used to recover from other conditions may allow also to recover from deadlock

Deadlock Prevention

- **Preventing mutual exclusion** – try to avoid non-sharable resources (every resource is made sharable and a process never needs to wait for any resource)
  - **Problem:** not very realistic since some resources are intrinsically non-sharable (for example, the access to a mutex lock cannot be shared by several processes)

- **Preventing hold and wait** – guarantee that whenever a process requests resources, it does not hold any other resources
  - Require process to request all resources before it begins execution (predicting future is hard, tend to over-estimate resources)
  - Alternatively, require process to release all its resources before it can request any additional resources
  - **Problem:** low resource utilization and starvation is possible

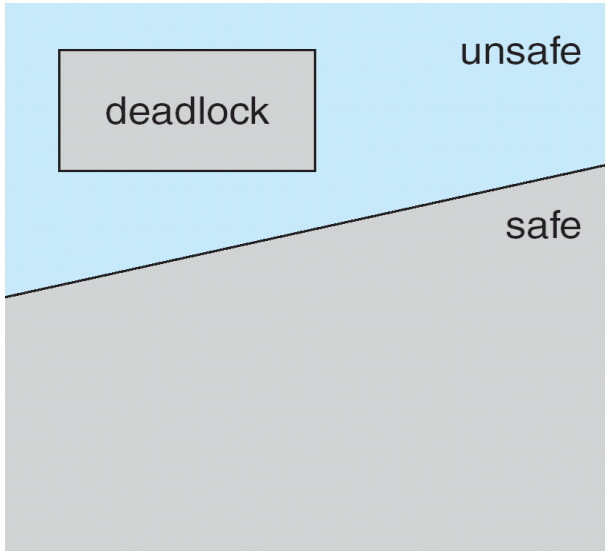
Deadlock Prevention

- **Preventing no preemption** – voluntarily release resources
  - If a process P fails to allocate some resources, we release all the resources that P is currently holding and, only when P can regain the old and the new resources that it is requesting, we restart it
  - Alternatively, if a process P fails to allocate some resources, we check whether they are allocated to some other process Q that is waiting for additional resources and, if so, we preempt the desired resources from Q and allocate them to the requesting process P
  - **Problem:** cannot generally be applied to resources such as mutex locks and semaphores
- **Preventing circular wait** – impose a total ordering of all resource types and require that each process requests resources in an increasing order of enumeration
  - **Problem:** programmers can write programs not following the ordering

Deadlock Avoidance

- Deadlock avoidance methods require **additional a priori information**, concerning which resources a process will use during its lifetime
  - Simplest and most useful method requires that each process declare the maximum number of resources of each type that it may need

- When a process requests an available resource, we must decide if its immediate allocation leaves the system in a safe state
  - **Safe state ⇒ no deadlock**
  - **Unsafe state ⇒ possibility of deadlock**



Safe State

- A system is in safe state if there **exists a safe sequence** $<P_1, P_2, \ldots, P_N>$ such that, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j (with $j < i$)

- Avoidance algorithms:
  - If a **single instance** of a resource type – **Resource allocation graph**
  - If **multiple instances** of a resource type – **Bankers algorithm**

Is This a Safe State?

- Consider that P_1 has made a request for a resource instance that, if fulfilled, leads the system to the following new state:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Is the system in a safe state?
  - Yes, sequence < P_1, P_3, P_4, P_0, P_2> satisfies the safety requirement!
  - Hence, we can grant P_1's request

Deadlock Detection

- If no deadlock prevention or avoidance method exists, then a deadlock situation may occur. In this environment, the system may:
  - Periodically invoke an algorithm that examines the state of the system and determines whether a deadlock has occurred
  - Provide an algorithm to recover from the deadlock
- Invoking a deadlock detection algorithm:
  - **For every resource request**, will incur in a considerable overhead
  - **Arbitrarily**, might lead the resource graph to contain many cycles, making it impossible to tell which of the deadlocked processes caused the deadlock

Is This a Deadlock?

- Consider that the system's state is as follows:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Is the system in a deadlock situation?
  - Although we can reclaim the resources held by P_0, the available resources are not sufficient to fulfill the requests of the other processes
  - Thus, a deadlock exists, consisting of processes P_1, P_2, P_3 and P_4

Recovery from Deadlock

- To recover from deadlocks, we can **abort processes** using two methods:
  - **Abort all deadlocked processes** – the results of the partial computations made by such processes are lost and probably will have to be recomputed
  - **Abort one process at a time until the deadlock cycle is eliminated** – incurs considerable overhead, since the deadlock detection algorithm must be invoked after each process is aborted

- In which order should we choose to abort?
  - Priority of the process
  - How long the process has computed and how much longer to completion
  - Resources the process has used
  - Resources the process needs to complete
  - How many processes will need to be terminated

Recovery from Deadlock

- Alternatively, to recover from deadlocks, we can successively **preempt some resources** from processes and give them to other processes until the deadlock cycle is broken. Three issues need to be addressed:
  - **Selecting a victim** – try to minimize costs such as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed
  - **Rollback** – return process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback (abort process and restart it from beginning)
  - **Starvation** – same process may always be picked as victim leading to starvation and, thus, a common solution is to include the number of rollbacks in the cost factor