

Open Source OS

Module 1: Process Management – Part II Synchronization

Module 1: Process Management –

Part II Synchronziation

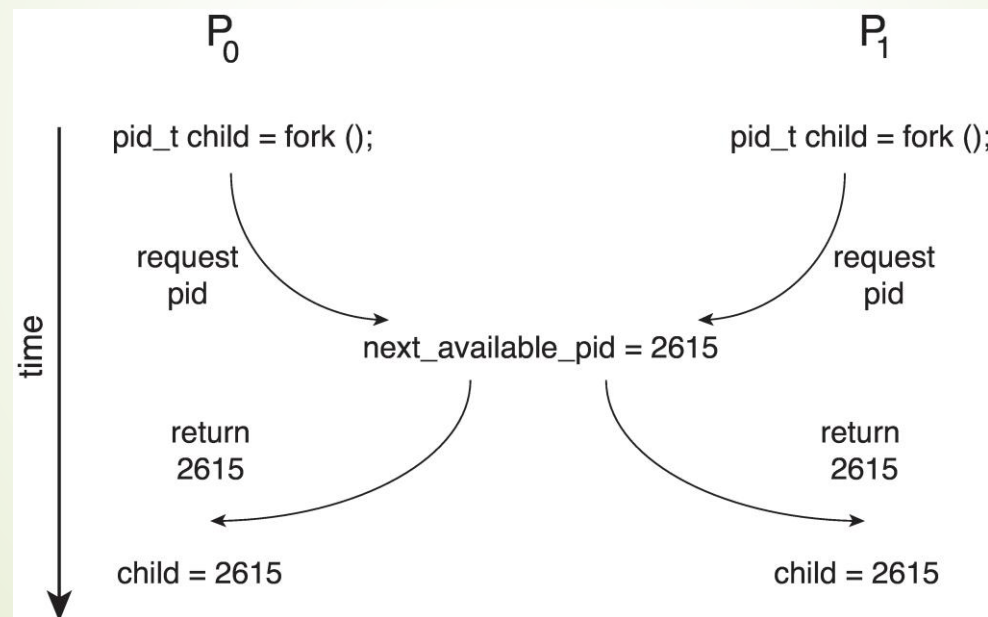
- Process Synchronization
 - The Critical-Section Problem
 - Peterson's Solution
 - Synchronization Tools: [Mutex Locks/ Semaphores/ Monitors](#)
- Classical Synchronization Problems: [bounded-buffer/readers-writers/dining-philosophers](#)
- Deadlocks
 - Deadlock Characterization
 - Methods for Handling Deadlocks: [Prevention/Avoidance/Detection/Recovery](#)

Background

- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining **data consistency** requires mechanisms to ensure the orderly execution of cooperating processes
 - We illustrated the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer, which leads to race condition

Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid`
 - the next available process identifier (pid)



- the variable `next_available_pid` could assign the same pid to two different processes!

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section**, a segment of code
 - that may be changing common variables, updating table, writing file, etc.
 - To avoid conflicts, no processes may be in critical section at the same time
- **Critical section problem** is to design protocol to solve this
 - Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

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

[Source: Operating Systems Concepts, 10th ed.]

Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
3. **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
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
 - What if the critical section contains code that runs for an hour?
 - Can some processes starve – never enter their critical sections?
 - What if there are two CPUs?

Software Solution 1

- Assume that the **load** and **store** machine-language instructions are **atomic**; that is, cannot be interrupted
- Two-process solution where the two processes share one variable:
 - **int turn;**
 - The variable **turn** indicates whose turn it is to enter the critical section

Algorithm for Process P_i

```
while (true) {
```

```
    turn = i;  
    while (turn == j)  
        ;
```

```
    /* critical section */
```

```
    turn = j;
```

```
    /* remainder section */
```

```
}
```

[Source: Operating Systems Concepts, 10th ed.]

Algorithms for Process P_i and P_j

```
while (true)
{
    turn = i;
    while (turn == j)
        ;
    /* critical section
    */
    turn = j;

    /* remainder
    section */
}
```

```
while (true)
{
    turn = j;
    while (turn == i)
        ;
    /* critical section
    */
    turn = i;

    /* remainder
    section */
}
```

Correctness of the Software Solution

- Mutual exclusion is preserved

P_i enters critical section only if:

turn = i

and **turn** cannot be both 0 and 1 at the same time

- What about the Progress requirement?
 - E.g. consider the case when P_j is faster than P_i
 - While P_i is waiting in the loop, P_j could be leaving (by setting $\text{turn}=i$) and entering again (by setting $\text{turn}=j$) **before** P_i could continue
- What about the Bounded-waiting requirement?

Peterson's Solution

- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- Two-process solution where the two processes share two variables:
 - **int turn;**
 - **boolean flag[2]**
 - The variable **turn** indicates whose turn it is to enter the critical section
 - The **flag** array is used to indicate if a process is **ready** to enter the critical section
 - **flag[i] = true** implies that process **P_i** is ready!

Algorithm for Process P_i

```
while (true) {
```

```
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j)  
        ;
```

```
    /* critical section */
```

```
    flag[i] = false;
```

```
    /* remainder section */
```

```
}
```

[Source: Operating Systems Concepts, 10th ed.]

Algorithms for Process P_i and P_j

```
while (true)
{
    flag[i] = true;
    turn = j;
    while (flag[j] && turn
        == j)
        ;
    /* critical section */
    flag[i] = false;

    /* remainder section */
}
```

```
while (true)
{
    flag[j] = true;
    turn = i;
    while (flag[i] && turn
        == i)
        ;
    /* critical section */
    flag[j] = false;

    /* remainder section */
}
```


Correctness of Peterson's Solution

➤ Provable that the three CS requirements are met:

1. Mutual exclusion is preserved

P_i enters CS only if:

either **flag[j]=false** or **turn=i**

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met

Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is **not** guaranteed to work on modern architectures
 - To improve performance, processors and/or compilers may **reorder** operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions
 - For single-threaded, this is ok as the result will always be the same
 - For multithreaded, the reordering may produce inconsistent or unexpected results!

Modern Architecture Example

- Two threads share the data:
`boolean flag = false;`
`int x = 0;`
- Thread 1 performs
`while (!flag)`
`;`
`print x`
- Thread 2 performs
`x = 100;`
`flag = true`
- What is the expected output?
100

[Source: Operating Systems Concepts, 10th ed.]

Modern Architecture Example (Cont.)

- However, since the variables `flag` and `x` are independent of each other, the instructions:

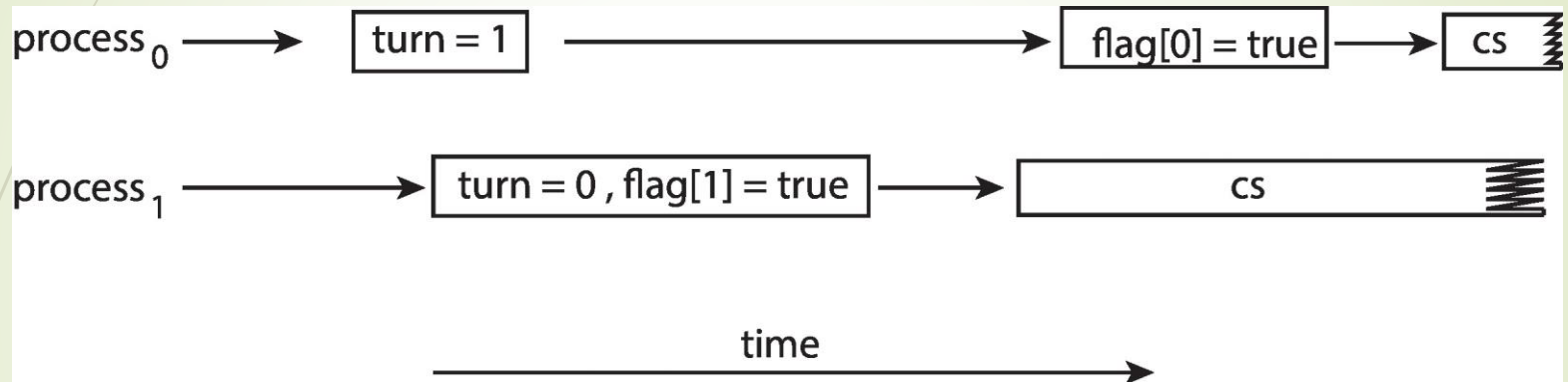
```
flag = true;  
x = 100;
```

for Thread 2 may be reordered

- If this occurs, the output may be 0!

Peterson's Solution Revisited

- The effects of **instruction reordering** in Peterson's Solution



- This allows both processes to be in their critical sections at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**

[Source: Operating Systems Concepts, 10th ed.]

Effect of Instruction Reordering on Peterson's Solution

```
while (true)
{
    turn = 1;
    flag[0] = true;
    while (flag[1] && turn
        == 1)
        ;
    /* critical section */
    flag[0] = false;

    /* remainder section */
}
```

```
while (true)
{
    turn = 0;
    flag[1] = true;
    while (flag[0] && turn
        == 0)
        ;
    /* critical section */
    flag[1] = false;


    /* remainder section */
}
```

Memory Barrier

- **Memory model** are the memory guarantees a computer architecture makes to application programs
- Memory models may be either:
 - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors
 - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors

Memory Barrier Instructions



- When a memory barrier instruction is performed, the system ensures that all loads and stores are **completed** before any subsequent load or store operations are performed
 - Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed
- 

Memory Barrier Example

- Returning to the example of slide 6.19
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x
```
- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```
- For Thread 1, we are guaranteed that the value of `flag` is loaded **before** the value of `x`
- For Thread 2, we ensure that the assignment to `x` occurs **before** the assignment `flag`

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - Operating systems using this not broadly scalable
- Two common forms of hardware support:
 1. Hardware instructions
 2. Atomic variables

Solution Using test_and_set()

- Shared boolean variable **lock**, initialized to **false**
- Solution:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
    /* remainder section */  
} while (true);
```

- Does it solve the critical-section problem?

Solution using compare_and_swap

- Shared integer **lock** initialized to 0;
- Solution:

```
while (true)
{
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans
- For example:
 - Let **sequence** be an atomic variable
 - Let **increment()** be operation on the atomic variable **sequence**
 - The Command:
increment(&sequence) ;
ensures **sequence** is incremented without interruption

Atomic Variables

- The **increment()** function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int t;
    do
    {
        t = *v;
    }
    while (t != compare_and_swap(v, t, t+1));
}
```




Synchronization Tools

Mutex Locks

- Hardware solutions are complicated
 - generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- The simplest: mutex lock
 - Boolean variable indicating if lock is available or not
 - First **acquire()** a lock
 - Then **release()** the lock
 - Calls to **acquire()** and **release()** must be **atomic**
 - Usually implemented via hardware atomic instructions such as compare-and-swap
- But this solution requires **busy waiting**
 - This lock therefore called a **spinlock**

Solution to CS Problem Using Mutex Locks

```
while (true) {  
    acquire lock  
  
    critical section  
  
    release lock  
  
    remainder section  
}
```

[Source: Operating Systems Concepts, 10th ed.]

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities
 - Semaphore **S** – integer variable
 - Can only be accessed via two indivisible (atomic) operations **wait()** and **signal()**
 - Originally called **P()** and **V()**
- Definition of the **wait()** operation

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

- Definition of the **signal()** operation

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

Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
 - Can implement a counting semaphore **S** as a binary semaphore
- With semaphores we can solve various synchronization problems

Semaphore Usage Example

Solution to the CS Problem

- Create a semaphore "**mutex**" initialized to 1

wait(mutex) ;

CS

signal(mutex) ;

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2

- Create a semaphore "**sync**" initialized to 0

P1 :

$S_1 ;$

signal(sync) ;

P2 :


wait(sync) ;

$S_2 ;$

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
 - Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
- Could have **busy waiting** in critical section implementation
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- 
- ▶ With each semaphore there is an associated waiting queue
 - ▶ Each entry in a waiting queue has two data items:
 - ▶ Value (of type integer)
 - ▶ Pointer to next record in the list
 - ▶ Two operations:
 - ▶ **block** – place the process on the appropriate waiting queue
 - ▶ **wakeup** – remove one of processes in the waiting queue and put it in the ready queue

Implementation with no Busy waiting (Cont.)

➤ Waiting queue

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```

[Source: Operating Systems Concepts, 10th ed.]

Problems with Semaphores

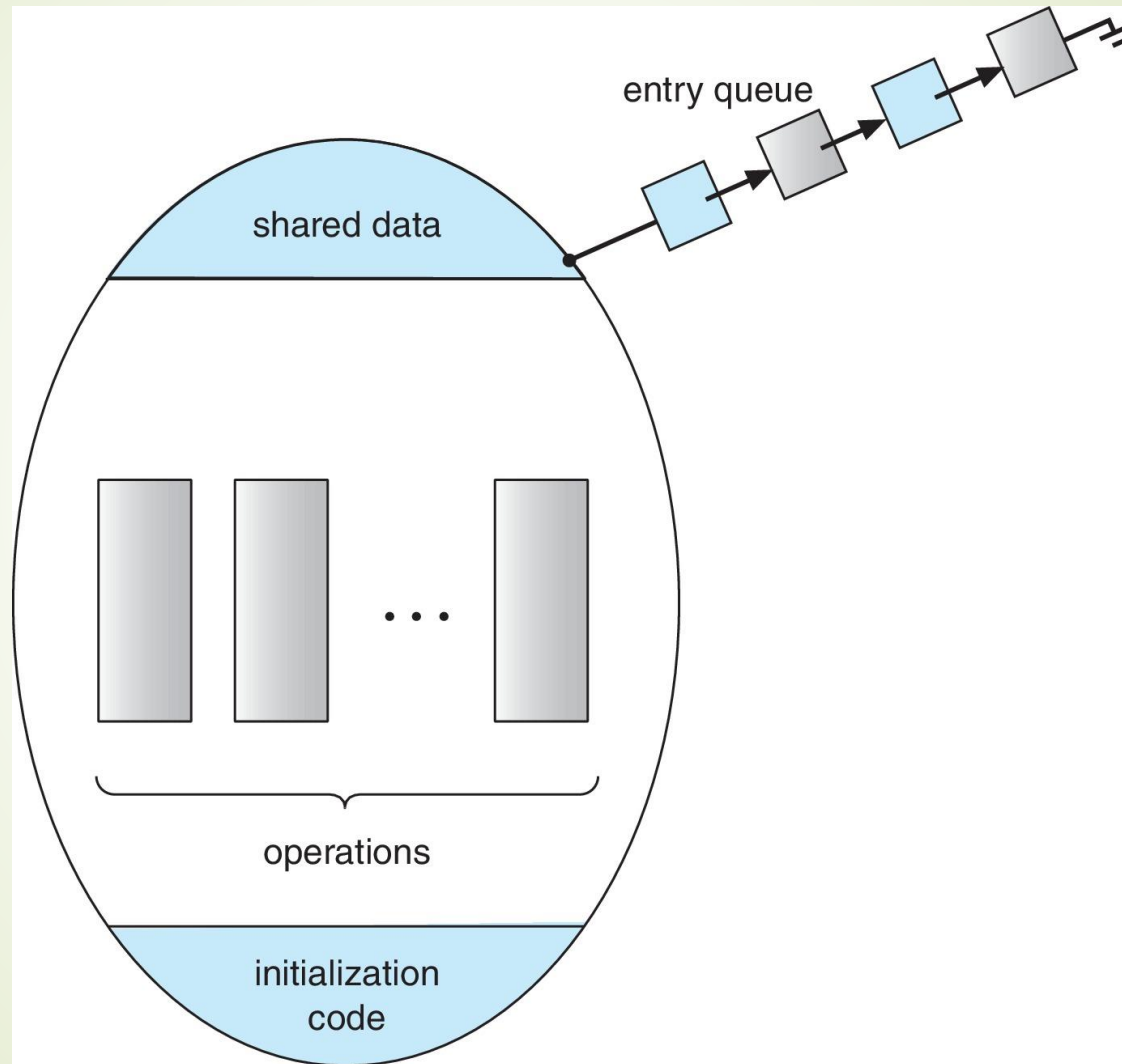
- Incorrect use of semaphore operations:
 - `signal(mutex) ... wait(mutex)`
 - `wait(mutex) ... wait(mutex)`
 - Omission of `wait(mutex)` and/or `signal(mutex)`
- Programmers are required to learn the correct use of semaphores and other synchronization tools

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
 - *Abstract data type*, internal variables only accessible by code within the procedure
 - Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }
    procedure P2 (...) { ... }
    procedure Pn (...) { ... }
    initialization code (...) { ... }
}
```

Schematic view of a Monitor



[Source: Operating Systems Concepts, 10th ed.]

Monitor Implementation Using Semaphores

Variables

```
semaphore mutex;  
mutex = 1;
```

Each procedure **P** is replaced by

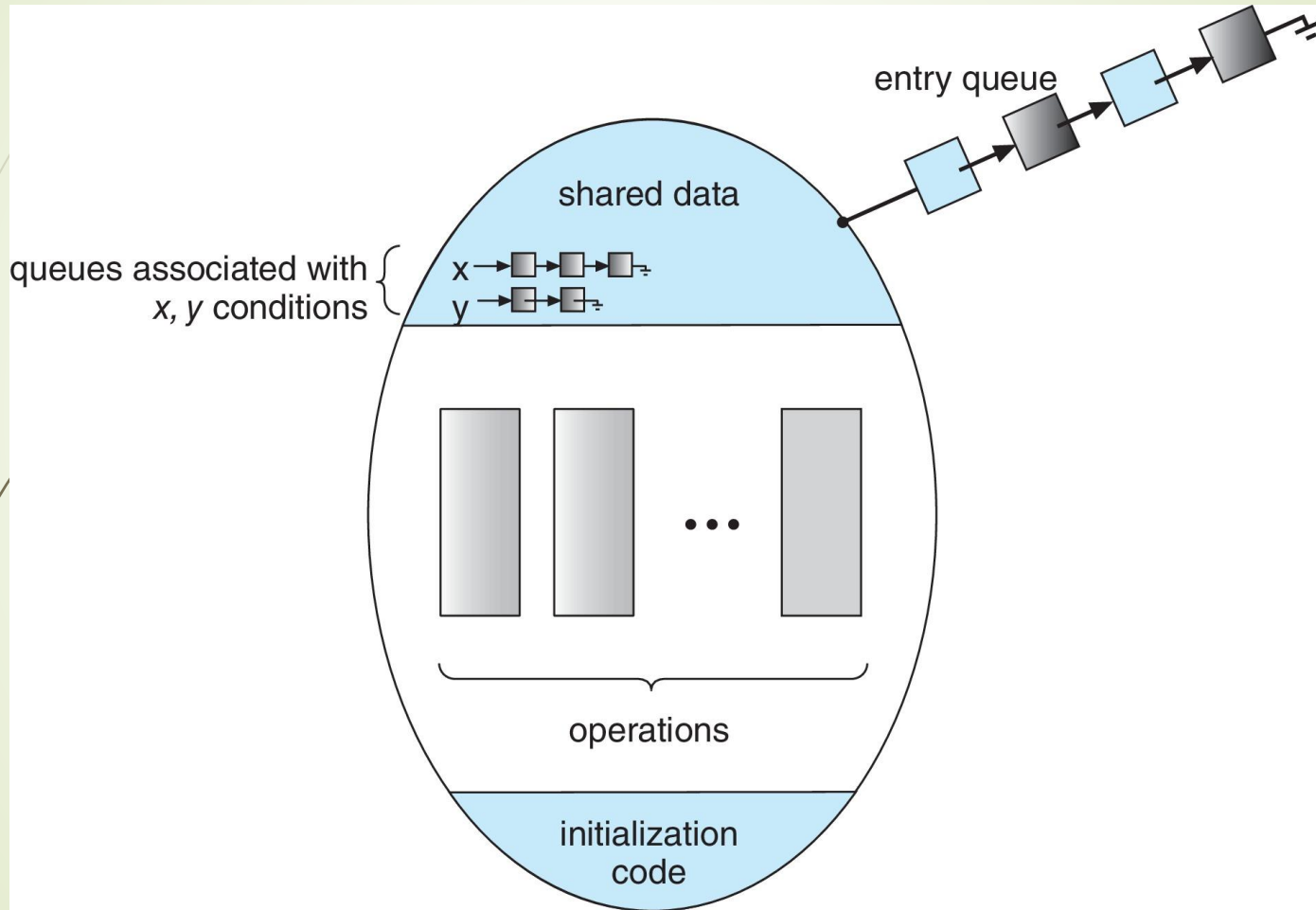
```
wait(mutex) ;  
...  
// body of P;  
...  
signal(mutex) ;
```

Mutual exclusion within a monitor is ensured

Condition Variables

- **condition x, y;**
- Two operations are allowed on a condition variable:
 - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
 - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
 - If no **x.wait()** on the variable, then it has no effect on the variable

Monitor with Condition Variables



[Source: Operating Systems Concepts, 10th ed.]

Example Usage of Condition Variable

Consider P_1 and P_2 that need to execute two statements S_1 and S_2 and that S_1 is required to happen **before** S_2

- Create a monitor with two procedures F_1 and F_2 that are invoked by P_1 and P_2 respectively
- One condition variable “x” initialized to 0
- One Boolean variable “done”

➤ **F1:**

```
S1;  
done = true;  
x.signal();
```



➤ **F2:**

```
if (done == false)  
    x.wait();  
S2;
```



Classical Synchronization Problems

Classical Problems of Synchronization

- 
- Classical problems used to test new synchronization schemes
 - Bounded-Buffer Problem
 - Readers and Writers Problem
 - Dining-Philosophers Problem
- 

Bounded-Buffer Problem

- n buffers, each can hold one item
- Semaphore **mutex** initialized to 1
- Semaphore **full** initialized to 0
- Semaphore **empty** initialized to n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

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

[Source: Operating Systems Concepts, 10th ed.]

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

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

[Source: Operating Systems Concepts, 10th ed.]

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
 - **Readers** – only read the data set; **no** updates
 - **Writers** – can both read and write
- Problem – allow multiple readers to read at the same time
 - Only one single writer can access the shared data at the same time
 - Several variations of how readers and writers are considered – with some form of priorities

Readers-Writers Problem (Cont.)

- Shared Data
 - Data set
 - Semaphore **rw_mutex** initialized to 1
 - Semaphore **mutex** initialized to 1
 - Integer **read_count** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process:

```
while (true)
{
    wait(rw_mutex) ;

    ...
    /* writing is performed */
    ...

    signal(rw_mutex) ;
}
```

[Source: Operating Systems Concepts, 10th ed.]

Readers-Writers Problem (Cont.)

- The structure of a reader process:

```
while (true){
    wait(mutex);
    read_count++;
    if (read_count == 1) /* first reader */
        wait(rw_mutex);

    signal(mutex);

    ...
    /* reading is performed */
    ...

    wait(mutex);
    read_count--;
    if (read_count == 0) /* last reader */
        signal(rw_mutex);

    signal(mutex);
}
```

[Source: Operating Systems Concepts, 10th ed.]

Readers-Writers Problem Variations

- The “**First** reader-writer” problem
 - A writer process might never write, if there’s still any readers reading
- The “**Second** reader-writer” problem:
 - Once a writer is ready to write, no “newly arrived reader” is allowed to read
- Both the first and second may result in starvation, leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

N philosophers sit at a round table with a bowl of rice in the middle



- They do not interact with their neighbors
- They spend their lives alternating thinking and eating
 - Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
 - Need both to eat, then release both when done
- In the case of 5 philosophers, the shared data
 - Bowl of rice (data set)
 - Semaphore chopstick [5] initialized to 1

[Source: Operating Systems Concepts, 10th ed.]

One Solution to Dining-Philosophers Problem

- The structure of Philosopher i :

```
while (true)
{
    wait (chopstick[i]);
    wait (chopstick[(i + 1) % 5]);

    /* eat for a while */

    signal (chopstick[i]);
    signal (chopstick[(i + 1) % 5]);

    /* think for a while */
}
```

- What is the problem with this algorithm?

[Source: Operating Systems Concepts, 10th ed.]

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
```

```
{  
    enum {THINKING, HUNGRY, EATING} state[5] ;  
    condition self[5];  
  
    void pickup (int i) {  
        state[i] = HUNGRY;  
        test(i);  
        if (state[i] != EATING) self[i].wait();  
    }  
  
    void putdown (int i) {  
        state[i] = THINKING;  
        // test left and right neighbors  
        test((i + 4) % 5);  
        test((i + 1) % 5);  
    }  
}
```

[Source: Operating Systems Concepts, 10th ed.]

Solution to Dining Philosophers (Cont.)

```
void test (int i)
{
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) )
    {
        state[i] = EATING;
        self[i].signal();
    }
}

initialization_code()
{
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
}
```

[Source: Operating Systems Concepts, 10th ed.]

Solution to Dining Philosophers (Cont.)

Each philosopher “i” invokes the operations **pickup()** and **putdown()** in the following sequence:

```
DiningPhilosophers.pickup(i);
```

```
/** EAT **/
```

```
DiningPhilosophers.putdown(i);
```

➡ No deadlock, but starvation is possible

Linux Synchronization

Linux:

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
 - Atomic integers
 - Mutex locks
 - Spinlocks, Semaphores
 - Reader-writer versions of both
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization

- Atomic variables
`atomic_t` is the type for atomic integer
- Consider the variables

```
atomic_t counter;  
int value;
```

<i>Atomic Operation</i>	<i>Effect</i>
<code>atomic_set(&counter, 5);</code>	<code>counter = 5</code>
<code>atomic_add(10, &counter);</code>	<code>counter = counter + 10</code>
<code>atomic_sub(4, &counter);</code>	<code>counter = counter - 4</code>
<code>atomic_inc(&counter);</code>	<code>counter = counter + 1</code>
<code>value = atomic_read(&counter);</code>	<code>value = 12</code>

POSIX Synchronization

- POSIX API provides
 - mutex locks
 - semaphores
 - condition variables
- Widely used on UNIX, Linux, and macOS

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

[Source: Operating Systems Concepts, 10th ed.]

POSIX Semaphores



- POSIX provides two versions of semaphores – **named** and **unnamed**
- Named semaphores can be used by unrelated processes
- Unnamed semaphores can be used only by threads in the same process

POSIX Named Semaphores

- Creating and initializing the named semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

[Source: Operating Systems Concepts, 10th ed.]

POSIX Unnamed Semaphores

- Creating and initializing the unnamed semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

[Source: Operating Systems Concepts, 10th ed.]

POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion
- Creating and initializing the condition variable:

```
pthread_mutex_t mutex;  
pthread_cond_t cond_var;  
  
pthread_mutex_init(&mutex, NULL);  
pthread_cond_init(&cond_var, NULL);
```

[Source: Operating Systems Concepts, 10th ed.]

POSIX Condition Variables

- Thread waiting for the condition `a == b` to become true:

```
pthread_mutex_lock(&mutex);  
while (a != b)  
    pthread_cond_wait(&cond_var, &mutex);  
  
pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);  
a = b;  
pthread_cond_signal(&cond_var);  
pthread_mutex_unlock(&mutex);
```

[Source: Operating Systems Concepts, 10th ed.]



Deadlocks

System Model

- System consists of resources
 - Resource types R_1, R_2, \dots, R_m
 - *CPU cycles, memory space, I/O devices*
 - Each resource type R_i has W_i instances
 - Each process utilizes a resource as follows:
 - **request**
 - **use**
 - **release**

Deadlock with Semaphores

- Data:
 - A semaphore **s1** initialized to 1
 - A semaphore **s2** initialized to 1
- Two processes P1 and P2
- **P1 :**
 - `wait(s1)`
 - `wait(s2)`
- **P2 :**
 - `wait(s2)`
 - `wait(s1)`

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** 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 $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that:
 - P_0 is waiting for a resource held by P_1
 P_1 is waiting for a resource held by P_2, \dots
 P_{n-1} is waiting for a resource held by P_n ,
and P_n is waiting for a resource held by P_0

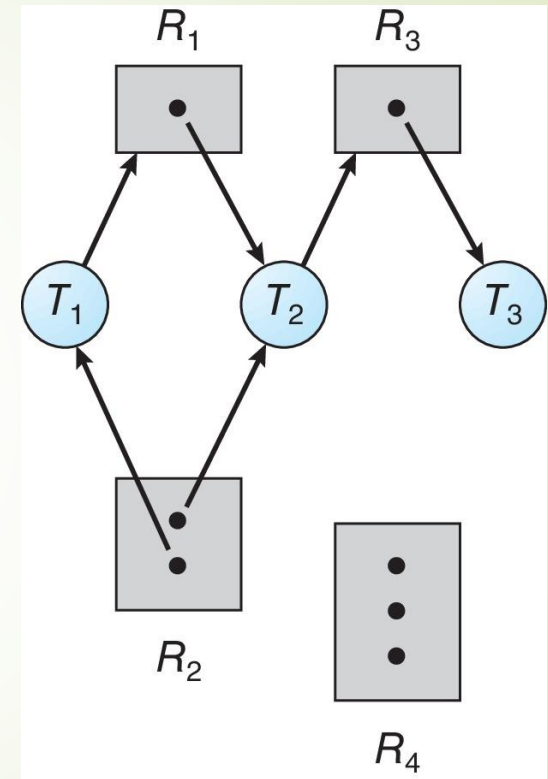
Resource-Allocation Graph

A set of vertices V and a set of edges E

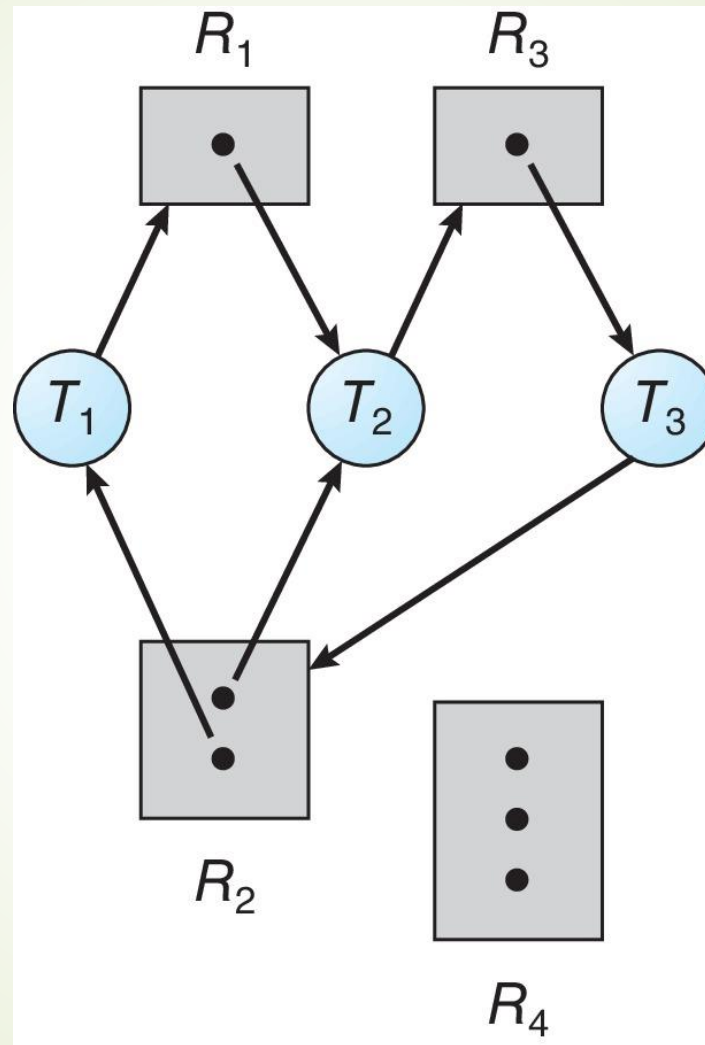
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph Example

- Resources
 - One instance of R1
 - Two instances of R2
 - One instance of R3
 - Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 holds one instance of R3

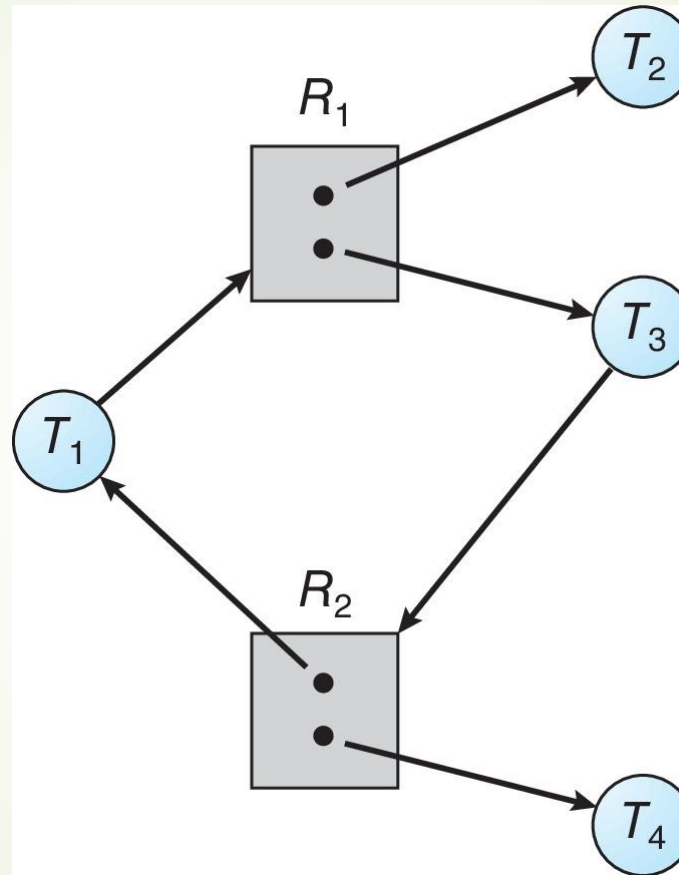


Resource Allocation Graph with a Deadlock



[Source: Operating Systems Concepts, 10th ed.]

Graph with a Cycle But no Deadlock



[Source: Operating Systems Concepts, 10th ed.]

Basic Facts

- If graph contains no cycles \Rightarrow no deadlock
- If graph contains a cycle \Rightarrow
 - if only one instance per resource type, then deadlock
 - if several instances per resource type, **possibility** of deadlock



Deadlock Handling Methods

Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
 - Deadlock prevention
 - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system

Deadlock Prevention

To **invalidate** one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Require process to request and allocate all its resources before it begins execution,
 - or allow process to request resources only when the process has not allocated any
 - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

➤ No Preemption:

- If a process that is holding some resources requests another resource that cannot be immediately allocated, then all resources currently being held are released
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, and the new ones that it is requesting

➤ Circular Wait:

- Impose a **total ordering** of all resource types, and require that each process requests resources in an increasing order of enumeration

Circular Wait

- Invalidating the circular wait condition is most common
 - Simply assign each resource (i.e., mutex locks) a unique number
- Resources must be acquired in order

➤ If:

```
first_mutex = 1  
second_mutex = 5
```

code for **thread_two** could not be written as follows:

```
/* thread_one runs in this function */  
void *do_work_one(void *param)  
{  
    pthread_mutex_lock(&first_mutex);  
    pthread_mutex_lock(&second_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&second_mutex);  
    pthread_mutex_unlock(&first_mutex);  
  
    pthread_exit(0);  
}  
  
/* thread_two runs in this function */  
void *do_work_two(void *param)  
{  
    pthread_mutex_lock(&second_mutex);  
    pthread_mutex_lock(&first_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&first_mutex);  
    pthread_mutex_unlock(&second_mutex);  
  
    pthread_exit(0);  
}
```

[Source: Operating Systems Concepts, 10th ed.]

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declares the ***maximum need*** of resources of each type
- The deadlock-avoidance algorithm **dynamically** examines the **resource-allocation state** to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests a resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL processes in the system such that: for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$
- That is:
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
 - When P_j finishes, P_i can obtain needed resources, execute, return allocated resources, and terminate
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state

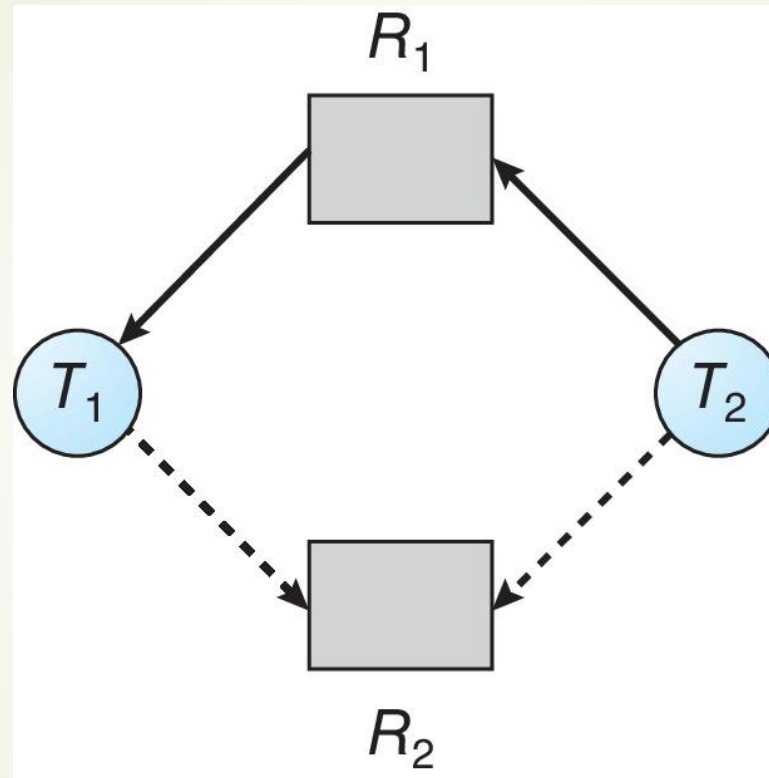
Avoidance Algorithms

- Single instance of a resource type
 - Use a resource-allocation graph
- Multiple instances of a resource type
 - Use the Banker's Algorithm

Resource-Allocation Graph Scheme

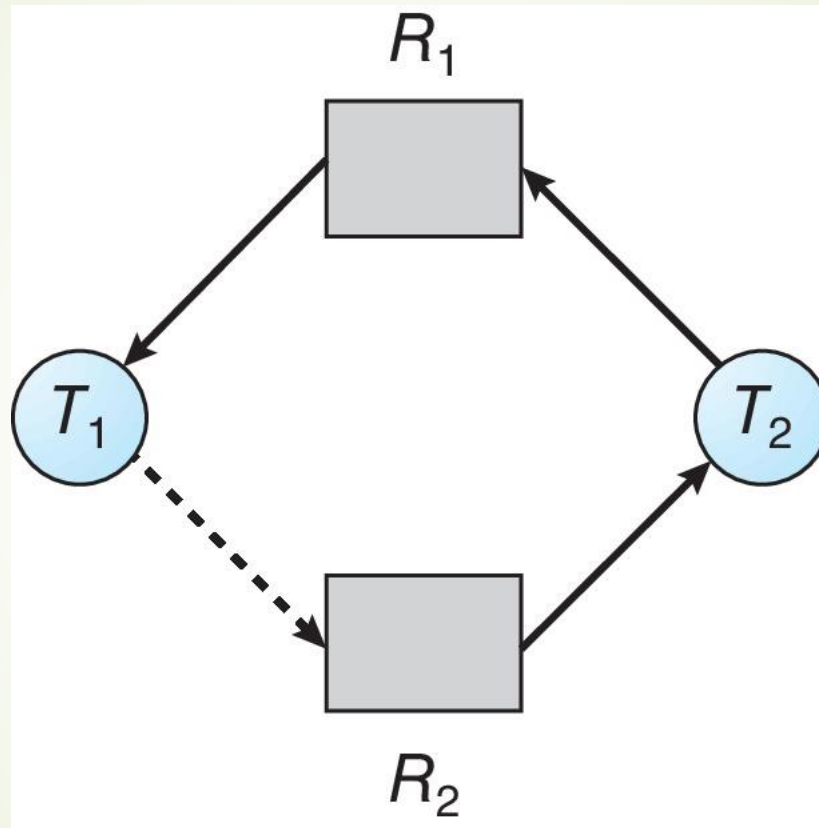
- **Claim edge** $P_i \rightarrow R_j$: (dashed line)
 - Process P_i may request resource R_j
 - Resources must be claimed *a priori* in the system
- Claim edge converts to request edge when a process requests a resource
- Request edge is converted to an assignment edge when the resource is allocated to the process
- Assignment edge reconverts to a claim edge, when the resource is released

Resource-Allocation Graph



[Source: Operating Systems Concepts, 10th ed.]

Unsafe State in Resource-Allocation Graph



[Source: Operating Systems Concepts, 10th ed.]

Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted **only if** converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Multiple instances of resources
- Each process must a priori claim maximum use
- When a process requests a resource, it may have to wait
- When a process gets all its resources, it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types

- **Available:** Vector of length m . If $Available[j] = k$, there are k instances of resource type R_j available
- **Max:** $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation:** $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is **currently allocated** k instances of R_j
- **Need:** $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Safety Algorithm

1. Let **Work** and **Finish** be vectors of length m and n , respectively. Initialize:

Work = Available

Finish [i] = false, for $i = 0, 1, \dots, n-1$

2. Find an i such that both:

(a) **Finish [i] = false**

(b) **Need_i ≤ Work**

If no such i exists, go to step 4

3. **Work = Work + Allocation_i**
Finish[i] = true
go to step 2

4. If **Finish [i] == true** for all i , then the system is in a safe state

[Source: Operating Systems Concepts, 10th ed.]

Resource-Request Algorithm for Process P_i

Request_i = request vector for process P_i . If **Request_i[j] = k** then process P_i wants **k** instances of resource type **R_j**

1. If **Request_i ≤ Need_i**, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If **Request_i ≤ Available**, go to step 3. Otherwise P_i must wait, since resources are not available
3. **Pretend** to allocate requested resources to P_i by modifying the state as follows:

$$\mathbf{Available} = \mathbf{Available} - \mathbf{Request}_i;$$

$$\mathbf{Allocation}_i = \mathbf{Allocation}_i + \mathbf{Request}_i;$$

$$\mathbf{Need}_i = \mathbf{Need}_i - \mathbf{Request}_i;$$

- If safe \Rightarrow the resources are allocated to P_i
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes: P_0 through P_4

3 resource types:

A (10 instances), B (5 instances), and C (7 instances)

- Snapshot at time T_0 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

[Source: Operating Systems Concepts, 10th ed.]

Example (Cont.)

- The content of the matrix **Need** is defined to be **Max – Allocation**

	<u>Need</u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria

[Source: Operating Systems Concepts, 10th ed.]

Example: P_1 Request (1,0,2)

Check that Request \leq Available: (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true)

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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	

- ▶ Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement
- ▶ Can request for (3,3,0) by P_4 be granted?
- ▶ Can request for (0,2,0) by P_0 be granted?

[Source: Operating Systems Concepts, 10th ed.]

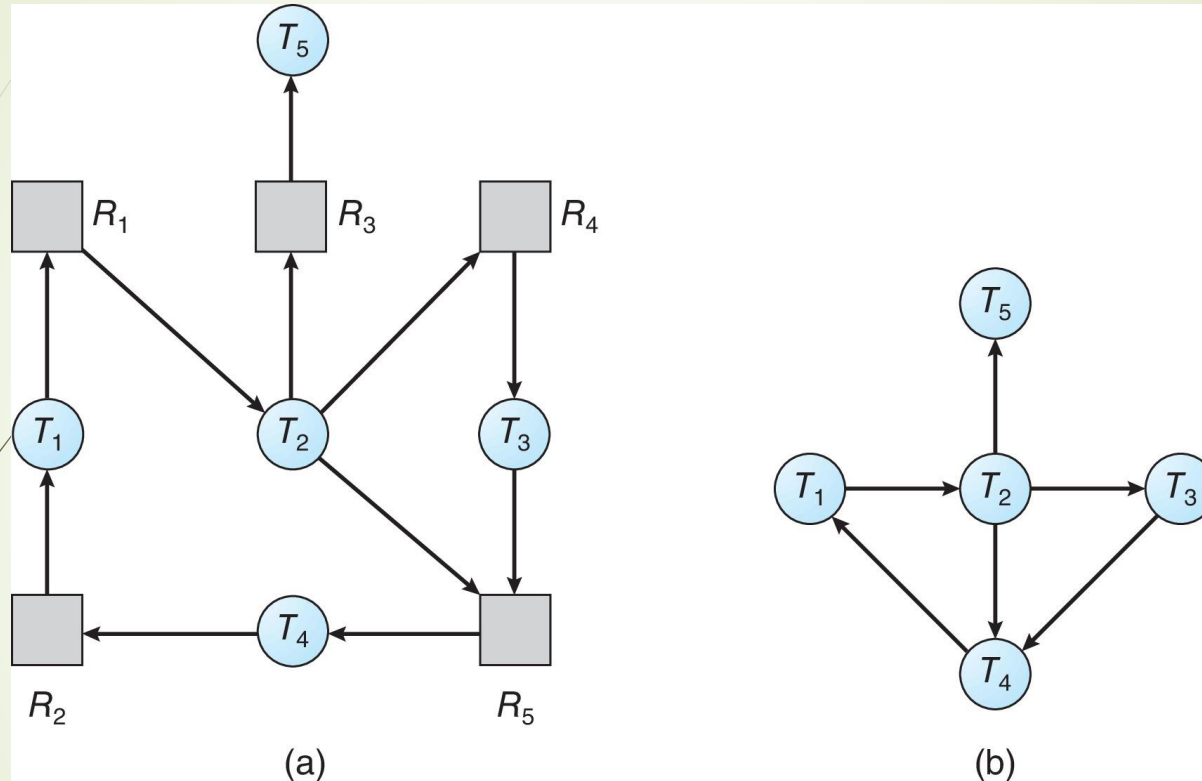
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain **wait-for** graph
 - Only one type of nodes: processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a **cycle** in the graph
 - If there is a cycle, there exists a deadlock
 - Cycle-detection algorithm in a graph requires an order of $O(n^2)$ operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

[Source: Operating Systems Concepts, 10th ed.]

Several Instances of a Resource Type

- **Available:** A vector of length m indicates the number of available resources of each type
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process
- **Request:** An $n \times m$ matrix indicates the current request of each process. If **Request** $[i][j] = k$, then process P_i is requesting k more instances of resource type R_j

Detection Algorithm

1. Let **Work** and **Finish** be vectors of length **m** and **n**, respectively. Initialize:
 - a) **Work** = **Available**
 - b) For $i = 1, 2, \dots, n$, if **Allocation_i** $\neq 0$, then **Finish[i] = false**; otherwise, **Finish[i] = true**
2. Find an index **i** such that both:
 - a) **Finish[i] == false**
 - b) **Request_i** \leq **Work**

If no such **i** exists, go to step 4

Detection Algorithm (Cont.)

3. **Work = Work + Allocation;**
Finish[i] = true
go to step 2

4. If **Finish[i] == false**, for some **i**, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if **Finish[i] == false**, then **P_i** is deadlocked

Algorithm requires an order of **$O(m \times n^2)$** operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0 :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	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 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in ***Finish[i] = true*** for all i
[Source: Operating Systems Concepts, 10th ed.]

Example (Cont.)

- P_2 requests an additional instance of type **C**

	<u>Request</u>		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes' requests
 - Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4

[Source: Operating Systems Concepts, 10th ed.]

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
 1. Priority of the process
 2. How long process has computed, and how much longer to completion
 3. Resources the process has used
 4. Resources process needs to complete
 5. How many processes will need to be terminated
 6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, so we should include number of rollbacks in cost factor

End of Module 1 -- Part II