Operating Systems

Jinghui Zhong （钟竞辉）
Office： B3-515
Email： jinghuizhong@scut.edu.cn

Course Schedule

- **Chinese students:**

Experiment:10%; Presentation: 5%; Poster 5%;

Attendance & Homework: 10%;

Final exam (close-book): 70%

- **Oversea students:**

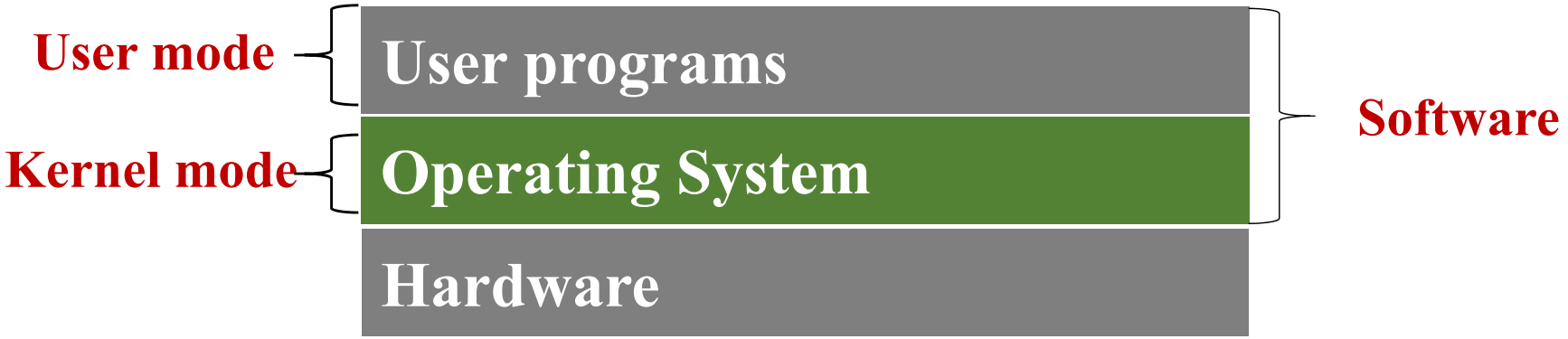
Experiment:20%; Presentation: 10%; Poster:10%

Homework & Attendance: 20%;

Final exam (close-book): 40%;

What is Operating System (OS)

- **A computer = Hardware + OS + User Programs**



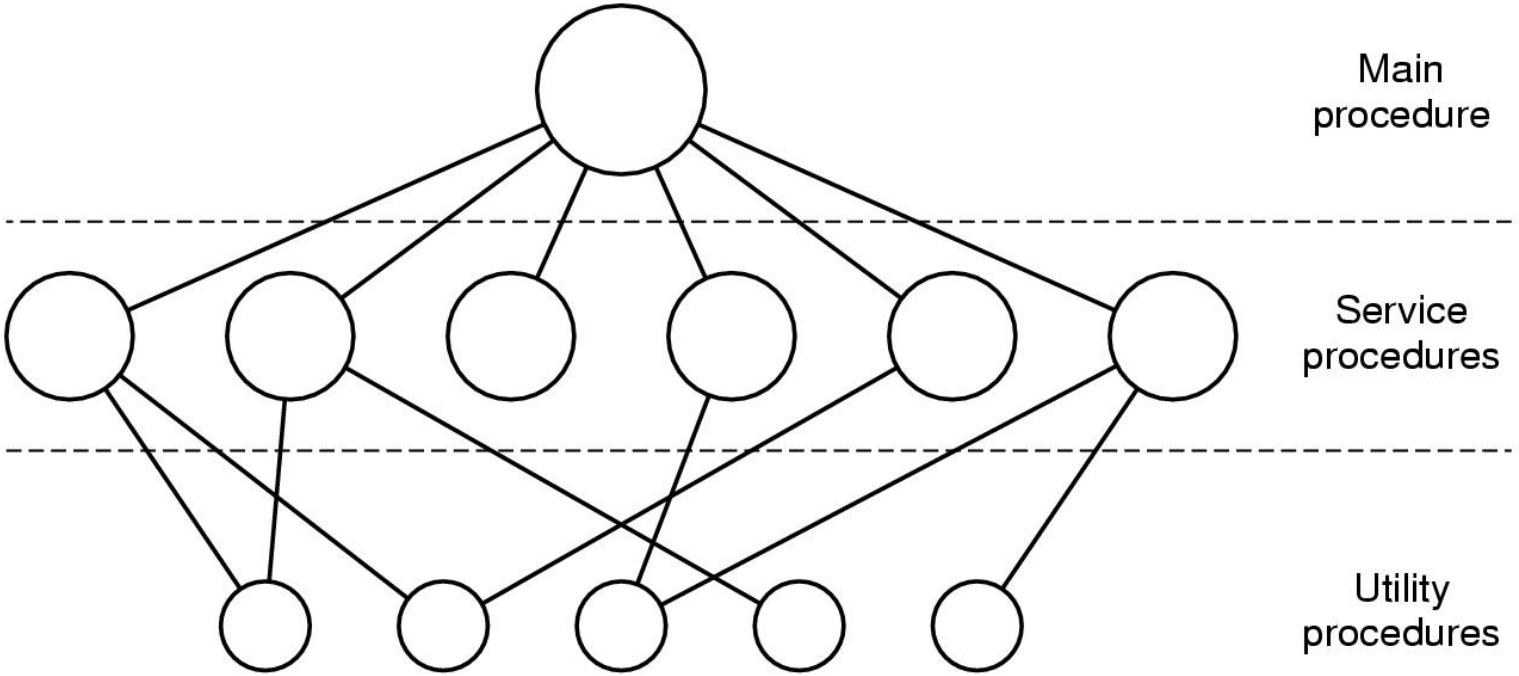
- **The distinct features of OS**
  - Run in kernel mode;
  - Has complete access to all hardware;
  - Can execute any instruction the machine is capable of executing;
  - Huge, complex, long-lived;

- **Two major tasks of OS.**

  To provide abstractions and to manage resources

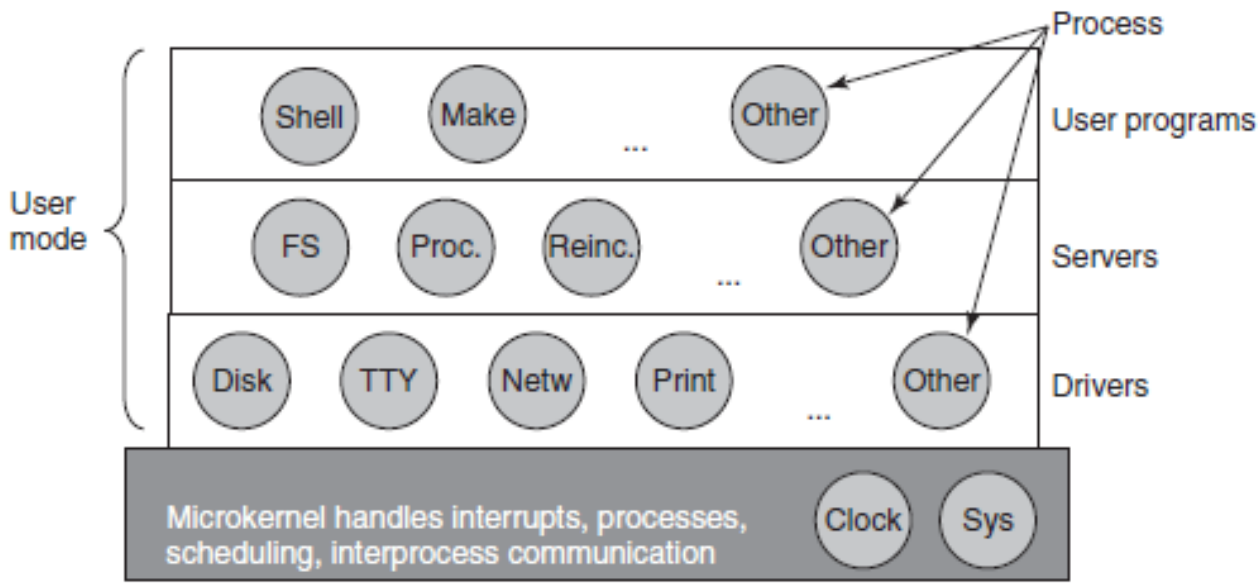
Monolithic System

- All operating system operations are put into a single file. The operating system is a collection of procedures, each of which can call any of the others. (e.g., Linux, windows)



Microkernels

- Split the OS into modules, only one runs in kernel mode and the rest run in user mode; (a lot communication)
- Put as little as possible into kernel model to improve the reliability of the system.
- Examples: MINIX 3



Processes

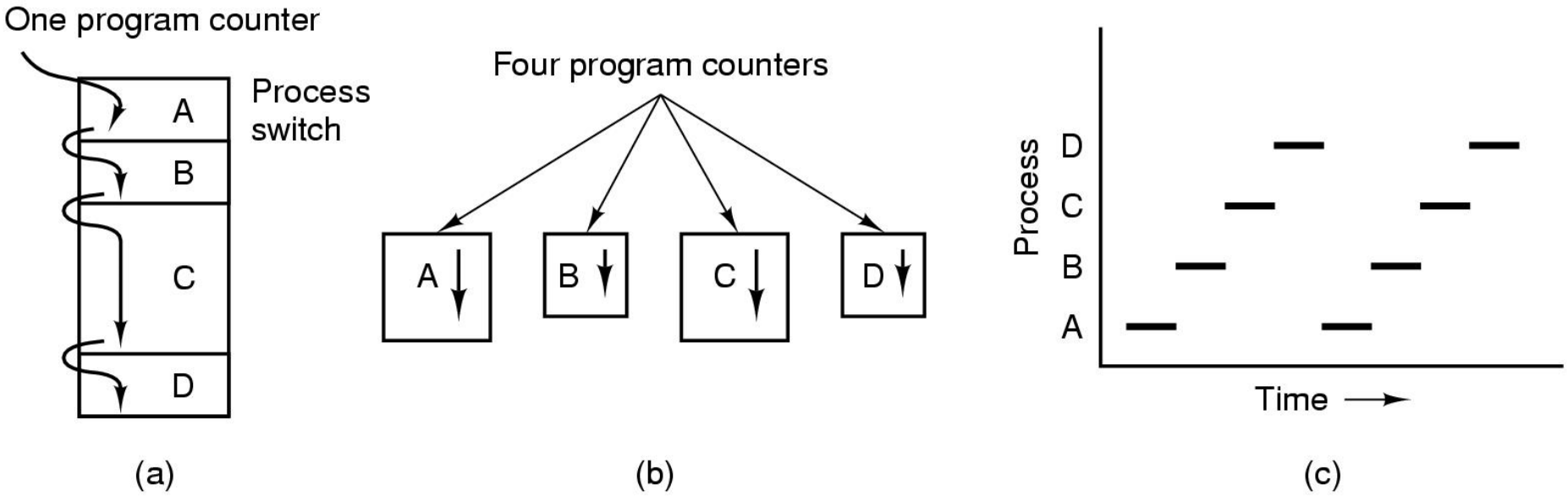
- **What is a process?**

① **a program in execution**.

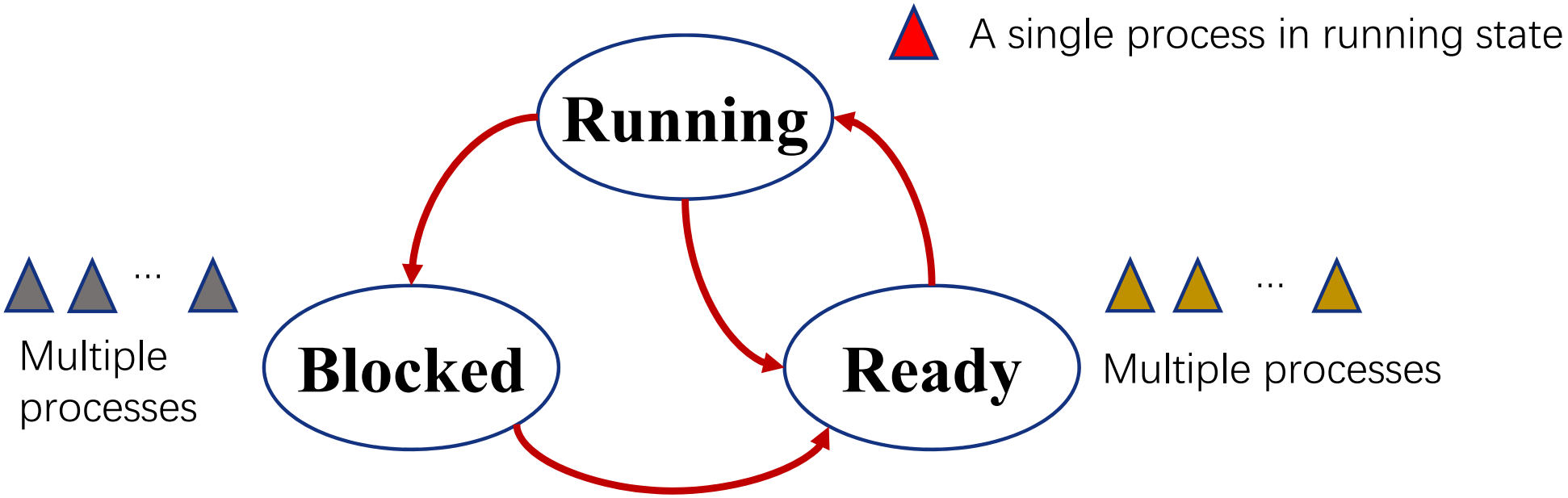
② **a container** that holds all the information needed to run a program.

Multiprogramming

- The CPU switches from process to process quickly, running each for tens or hundreds of milliseconds.
- At anytime, the CPU is running only one process.



Process States



- **Running：** using the CPU at that instant.
- **Ready：** runnable; temporarily stopped to let another process run.
- **Blocked：** unable to run until some external event happens.

Implementation of Processes

- The OS maintains a **Process Table** with one entry (called a **process control block (PCB)**) for each process.

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Modeling Multiprogramming

- Suppose a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once. The CPU utilization can be obtained by:

$$\text{CPU utilization} = 1 - p^n$$

Thread

●What is a thread?

A thread of execution is the smallest sequence of programmed instructions that **can be managed independently by a scheduler**, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases **a thread is a component of a process**. Multiple threads **can exist within one process, executing concurrently and sharing resources** such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Thread vs. Process

- A thread – **lightweight process,** a basic unit of CPU utilization.
- It comprises a thread ID, a program counter, a register set, and a stack.
- A traditional (**heavyweight**) process has a single thread of control.
- If the process has multiple threads of control, it can do more than one task at a time. This situation is called **multithreading**.
- Process： used to group resources together;

  Thread： the entity scheduled for execution on the CPU.

POSIX Threads

- **POSIX (Portable Operating System Interface)**
  is a set of standard operating system interfaces based on the Unix operating system.

The need for standardization arose because enterprises using computers wanted to be able to develop programs that could be moved among different manufacturer's computer systems directly.

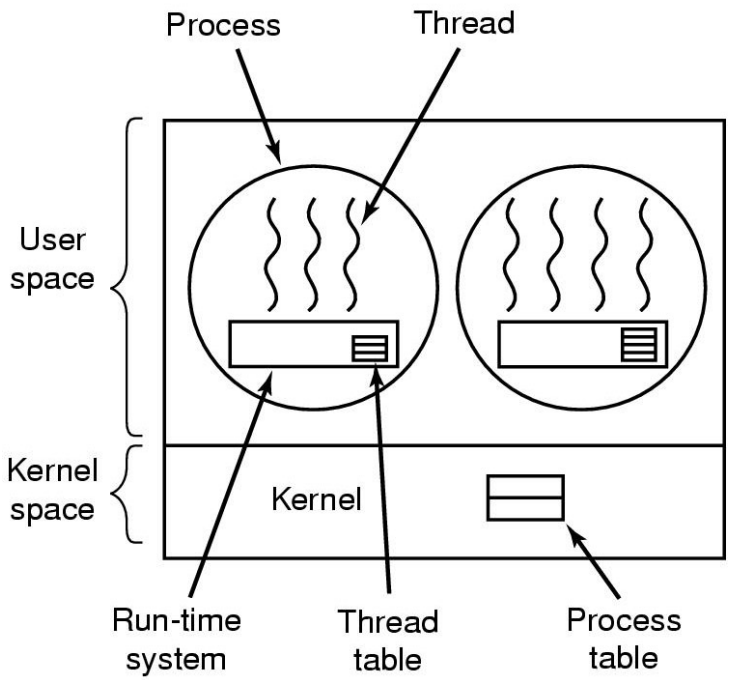
Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Implementing threads in user space

- Advantages: fast, flexible, scalable
- Drawbacks：

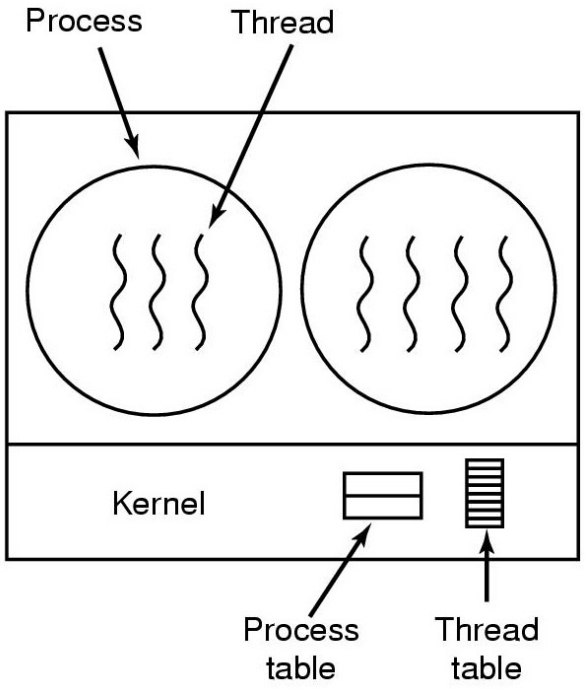
1) Blocking blocks all threads in a process;

2) No thread-level parallelism on multiprocessor.



Implementing threads in kernel

- Advantages: Blocking blocks only the appropriate thread in a process; Support thread-level parallelism on multiprocessor;
- Disadvantages: Slow thread management; large thread tables



Race Conditions

- **Race conditions:** situations in which several processes access shared data and the final result depends on the order of operations.

- With increasing parallelism due to increasing number of cores, race condition are becoming more common.

Critical Regions

- Key idea to avoid race condition： **prohibit more than one process from reading and writing the shared data at the same time.**

- **Critical Region:** part of the program where the share memory is accessed。

```
// 临界区结构对象
CRITICAL_SECTION g_cs;
// 共享资源
char g_cArray[10];
UINT ThreadProc10(LPVOID pParam)
{
    // 进入临界区
    EnterCriticalSection(&g_cs);
    // 对共享资源进行写入操作
    for (int i = 0; i < 10; i++)
    {
    g_cArray[i]  = a;
    Sleep(1);
    }
    // 离开临界区
    LeaveCriticalSection(&g_cs);
    return 0;
}
UINT ThreadProc11(LPVOID pParam)
{
    // 进入临界区
    EnterCriticalSection(&g_cs);
    // 对共享资源进行写入操作
    for (int i = 0; i < 10; i++)
    {
        g_cArray[10 - i - 1] = b;
        Sleep(1);
    }
    // 离开临界区
    LeaveCriticalSection(&g_cs);
    return 0;
}
……
void CSample08View::OnCriticalSection()
{
    // 初始化临界区
    InitializeCriticalSection(&g_cs);
    // 启动线程
    AfxBeginThread(ThreadProc10, NULL);
    AfxBeginThread(ThreadProc11, NULL);
    // 等待计算完毕
    Sleep(300);
    // 报告计算结果
    CString sResult = CString(g_cArray);
    AfxMessageBox(sResult);
}
```

Semaphores [E.W. Dijkstra, 1965]

- **A SEMAPHORE, S, is a structure consisting of two parts:**
  - **(a) an integer counter, COUNT**
  - **(b) a queue of pids of blocked processes, Q**

- That is,

```
struct sem_struct {
  int count;
  queue Q;
} semaphore;

semaphore S;
```

Semaphores [E.W. Dijkstra, 1965]

- There are two operations on semaphores, **UP** and **DOWN** (PV). These operations must be **executed atomically** (that is in mutual exclusion). Suppose that P is the process making the system call. The operations are defined as follows:

```
DOWN(S):

        if (S.count > 0)
                S.count = S.count - 1;
        else
                block(P); that is,
            (a) enqueue the pid of P in S.Q,
            (b) block process P (remove the pid from the ready queue)
            (c) pass control to the scheduler.
```

Semaphores [E.W. Dijkstra, 1965]

```
UP(S):

    if (S.Q is nonempty)
        wakeup(P) for some process P in S.Q; that is,
            (a) remove a pid from S.Q (the pid of P),
            (b) put the pid in the ready queue, and
            (c) pass control to the scheduler.
    else
        S.count = S.count + 1;
```

Mutual Exclusion Solution

- Semaphores do not require busy-waiting, instead they involve BLOCKING.

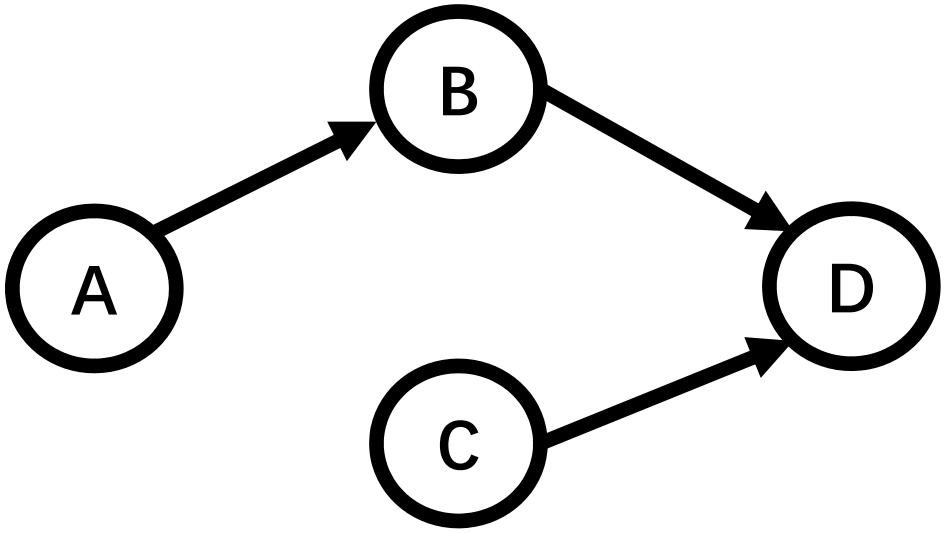
```
semaphore mutex = 1;   // set mutex.count = 1

DOWN(mutex);
      - critical section -
UP(mutex);
```

Using Semaphores

- **Process Synchronization:**

Suppose we have 4 processes: A, B, C, and D. A must finish executing before B start. B and C must finish executing before D starts.



How many semaphores should be used to achieve the above goal?

Using Semaphores

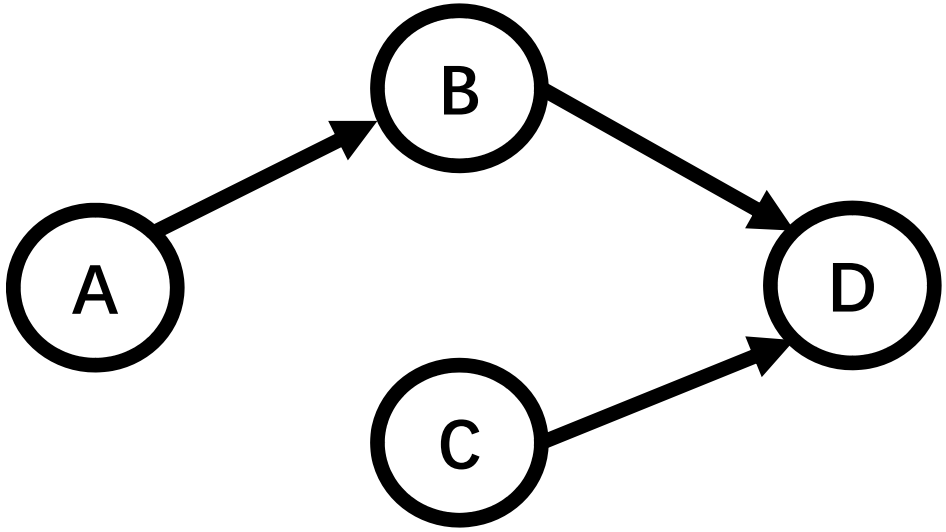
- Process Synchronization:

Process A:
- do work of A
UP(S1); /* Let B or C start */

Process B:
DOWN(S1); /* Block until A is finished */
- do work of B
UP(S2);

Process C:
- do work of C
UP(S3);

Process D:
DOWN(S2);
DOWN(S3);
- do work of D



Process Behavior

- Nearly all processes alternate bursts of computing with I/O requests.
- CPU-bound: Spend most of the time computing.
- IO-bound: Spend most of the time waiting for I/O.

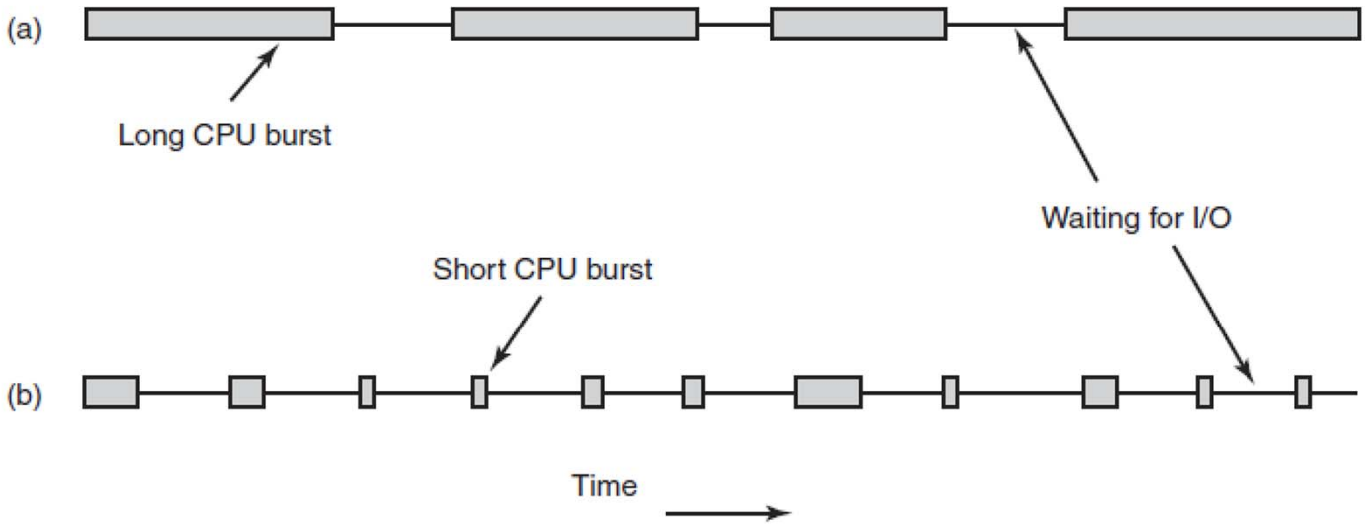


Figure 2-39. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Process Scheduling

- **Scheduler:** A part of the operating system that decides which process is to be run next.
- **Scheduling Algorithm:** a policy used by the scheduler to make that decision.
- To make sure that no process runs too long, a clock is used to cause a periodic interrupt (usually around 50-60 Hz); that is, about every 20 msec.
- **Preemptive Scheduling:** allows processes that are runnable to be temporarily suspended so that other processes can have a chance to use the CPU.

Scheduling Algorithm Goals

1. **Fairness** - each process gets its fair share of time with the CPU.
2. **Efficiency** - keep the CPU busy doing productive work.
3. **Response Time** - minimize the response time for interactive users.
4. **Turnaround Time** - minimize the average time from a batch job being submitted until it is completed.
5. **Throughput** - maximize the number of jobs processed per hour.

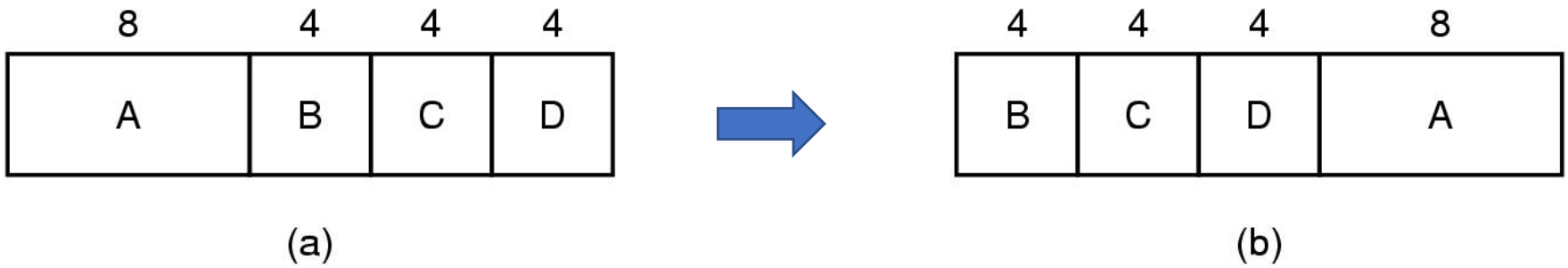
First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
- Waiting time for P_1, P_2, and P_3 = 0， 24， 27
- Average waiting time = (0+24+27)/3 = 17

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.



An example of shortest job first scheduling

- Drawback?

The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

Shortest-Job-First (SJR) Scheduling

- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until it completes its CPU burst.
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

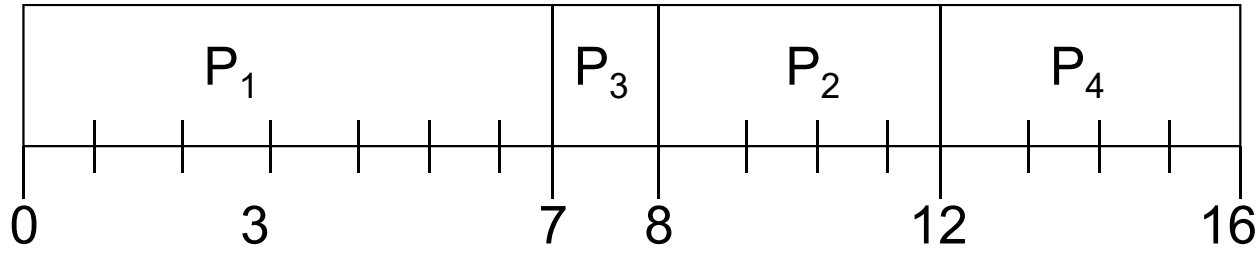
- SJF (non-preemptive)

Average waiting time = ?

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

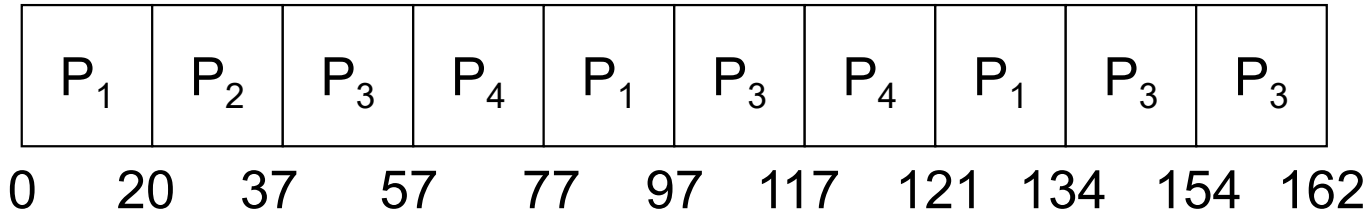
Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

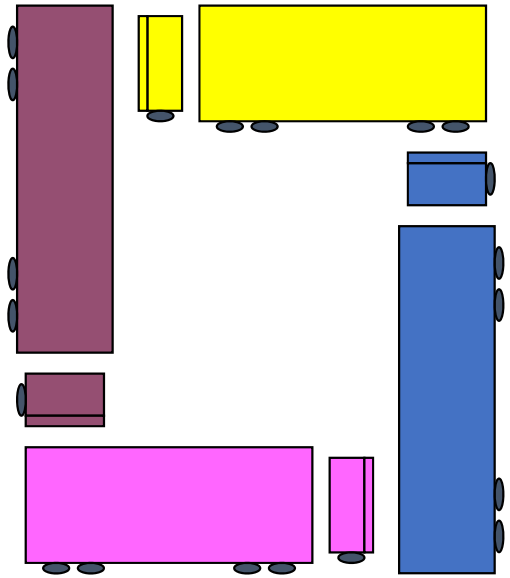


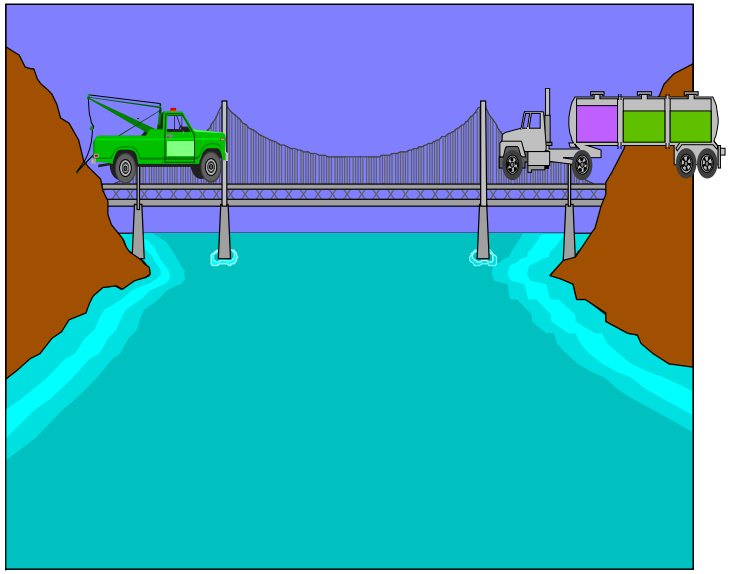
- Typically, higher average turnaround than SJF, but better *response*.

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem: Starvation – low priority processes may never execute.
- Solution: Aging – as time progresses increase the priority of the process.

Computer Deadlock: What it is and how to model it

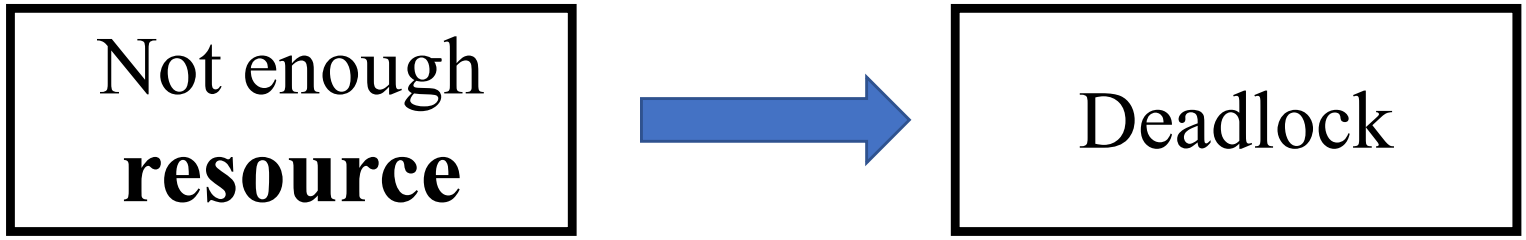




What is deadlock?

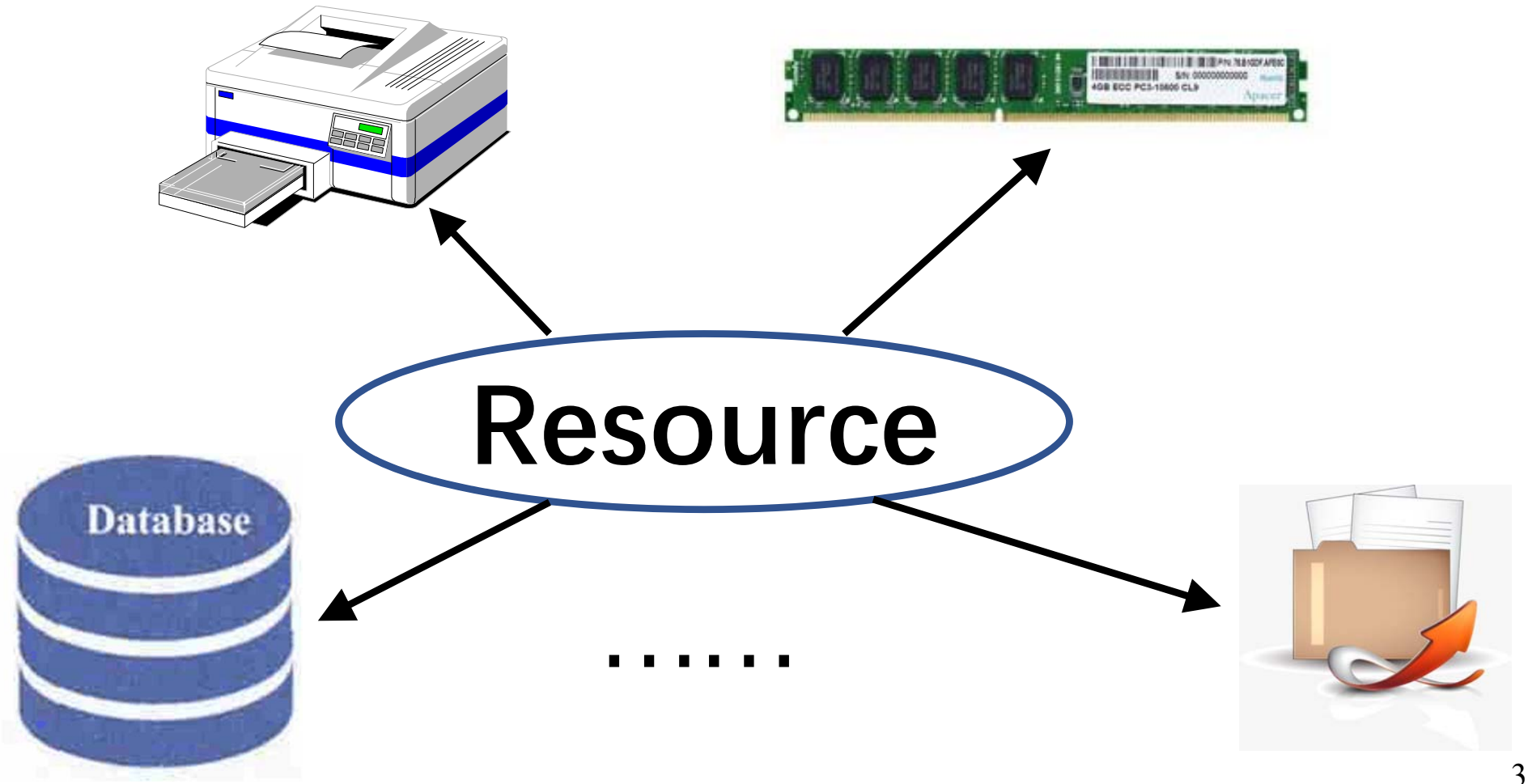
- **Definition**

A set of processes(e.g., ) is in a ***deadlock state*** when every process in the set is waiting for a **resource** that can only be released by another process in the set.



Resources

A **resource** is anything that can be acquired, used, and released over the course of time.



Two types of Resources

- **Preemptable resources**

  can be taken away from a process with no ill effects (e.g. memory)
- **Nonpreemptable resources**

  will cause the process to fail if taken away (e.g. CD recorder)

Potential deadlocks that involve Preemptable resources can usually be resolved by reallocating resources from one process to another.

Conditions for Deadlocks

Mutual exclusion

Resources are held by processes in a non-sharable (exclusive) mode.

Hold and Wait

A process holds a resource while waiting for another resource.

No Preemption

There is only voluntary release of a resource - nobody else can make a process give up a resource.

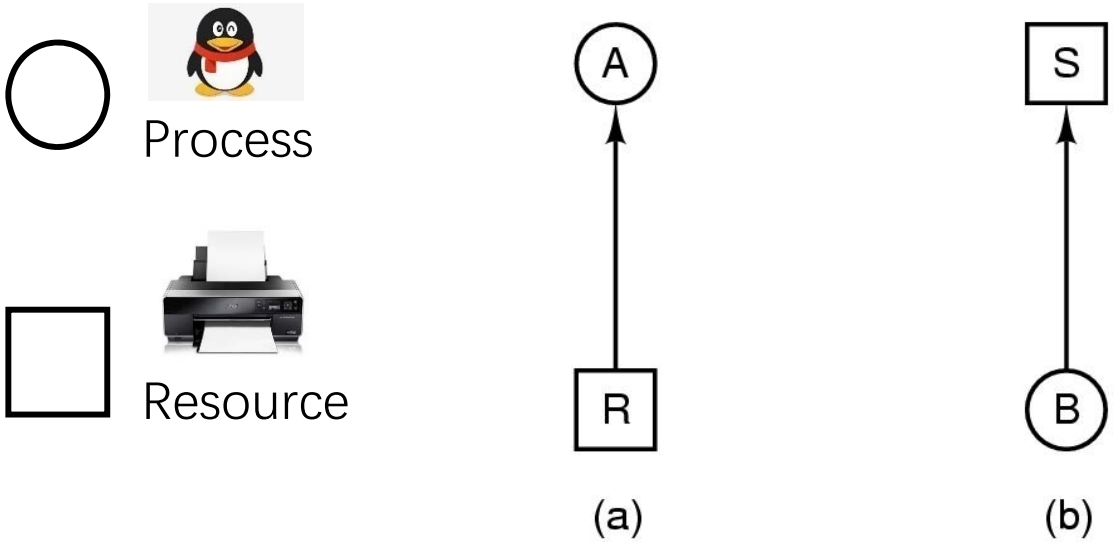
Circular Wait

Process A waits for Process B waits for Process C .... waits for Process A.

ALL of these four conditions **must** happen simultaneously for a deadlock to occur.

Deadlock Modeling
(Resource with single instance)

- Modeled with **directed graphs**

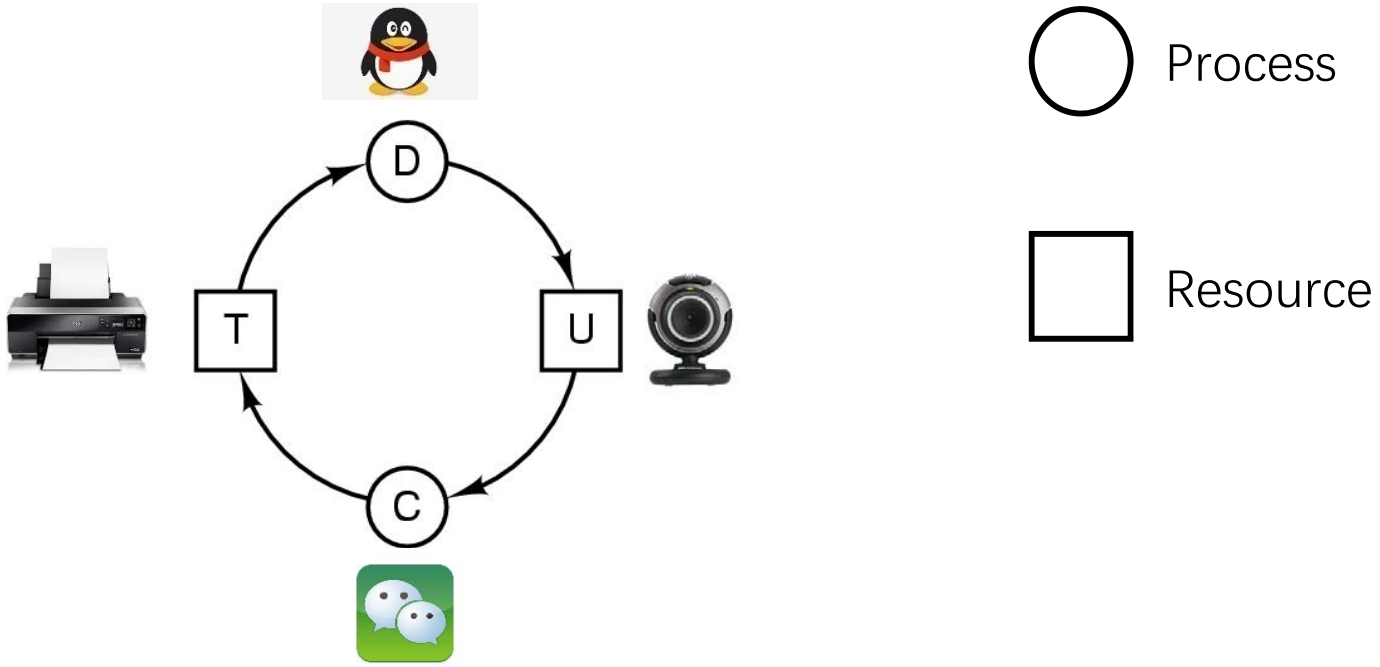


(a) resource R assigned to process A

(b) process B is requesting/waiting for resource S

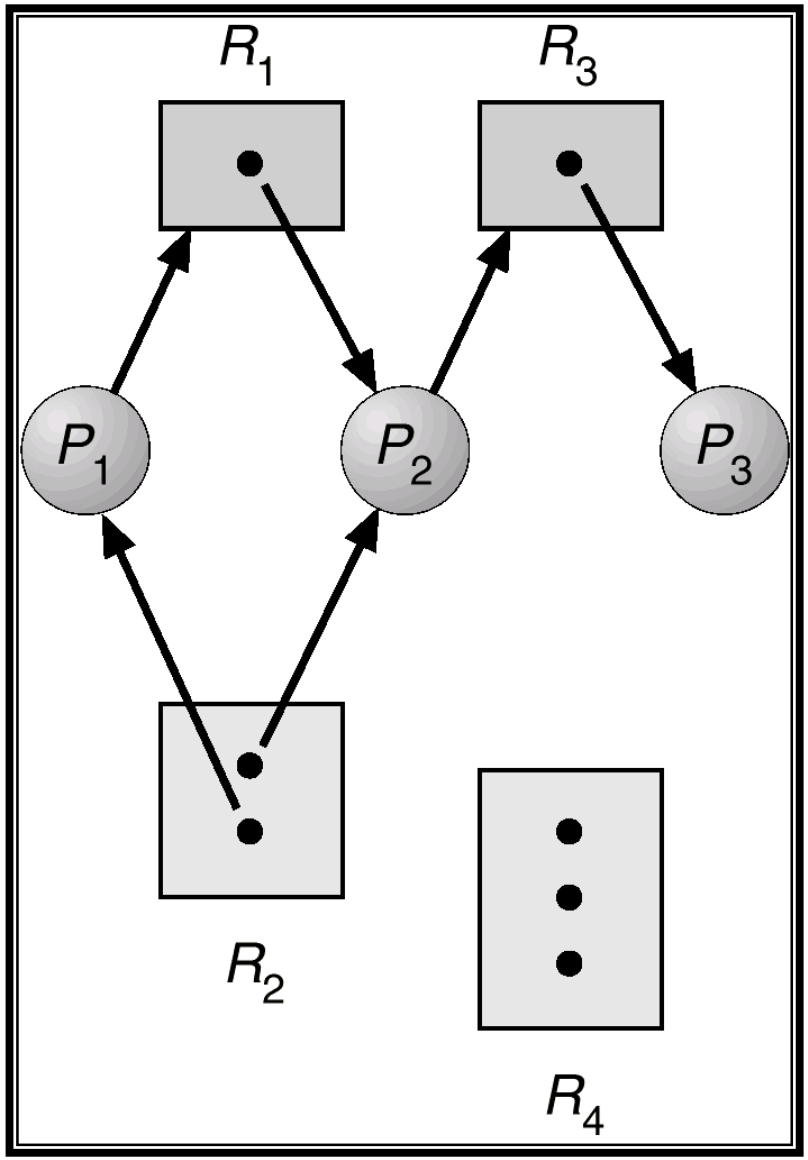
Deadlock Modeling
(Resource with single instance)

- Modeled with **directed graphs**



Process C and D are in deadlock over resources T and U

Example (1)



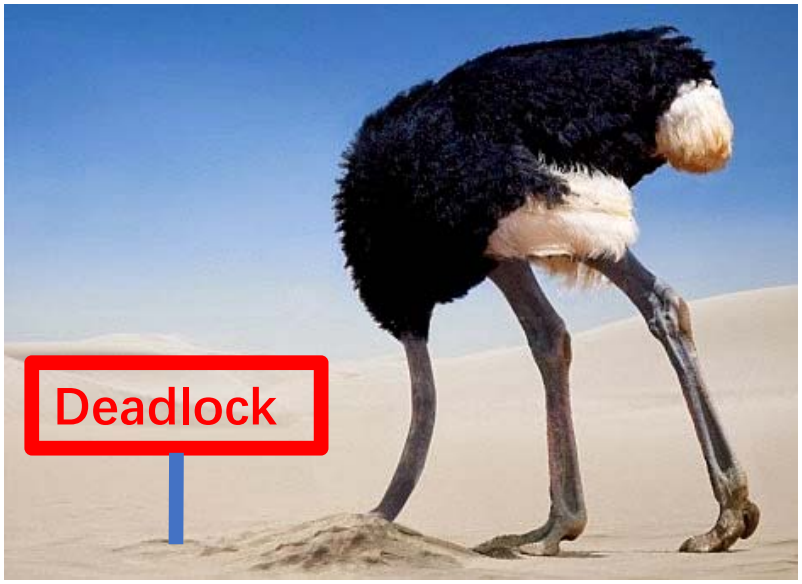
Basic Fact

- If graph contains **no cycles** ⇒ **no** deadlock.

- If graph contains **a cycle** ⇒
  - ✓ if only one instance per resource type, then deadlock.
  - ✓ if several instances per resource type, possibility of deadlock.

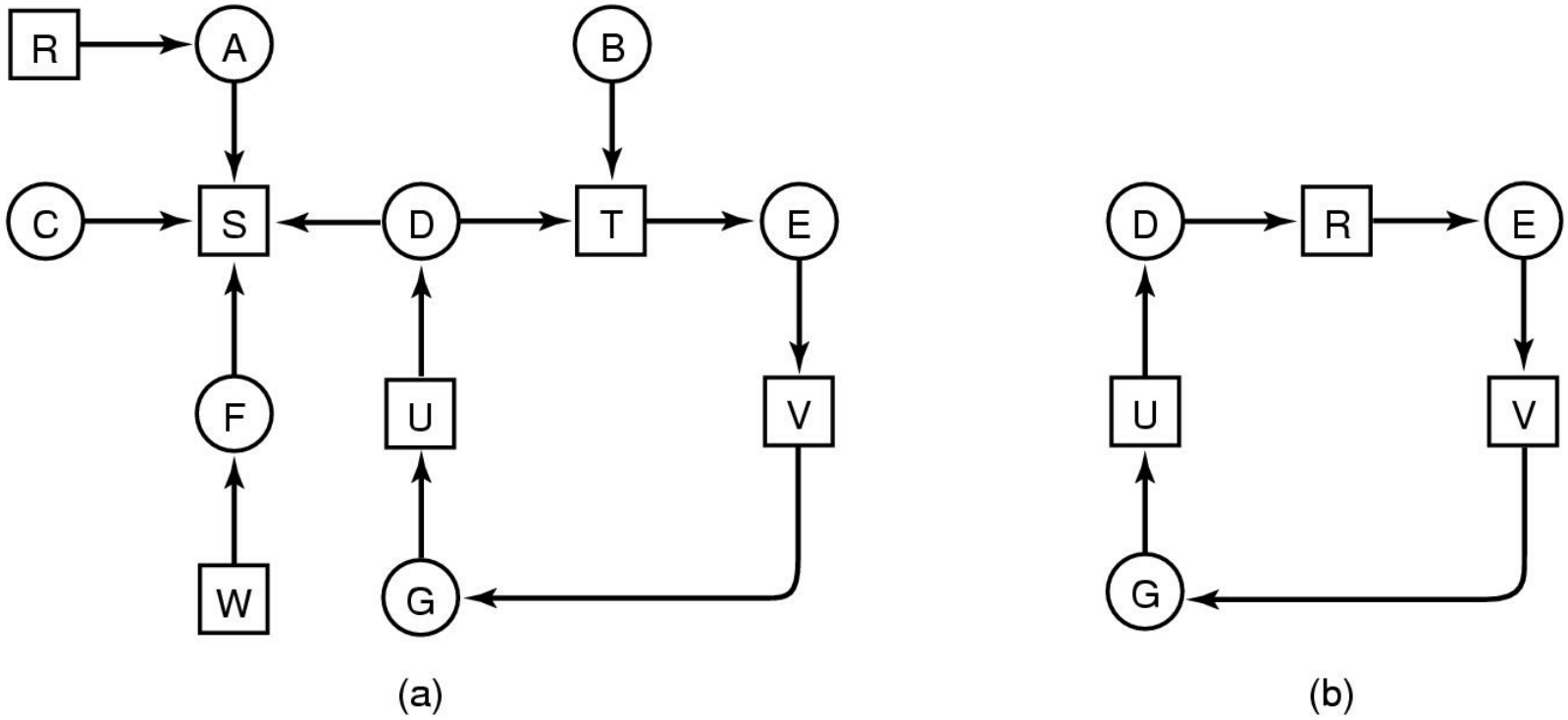
The Ostrich Algorithm

- Pretend that there is no problem



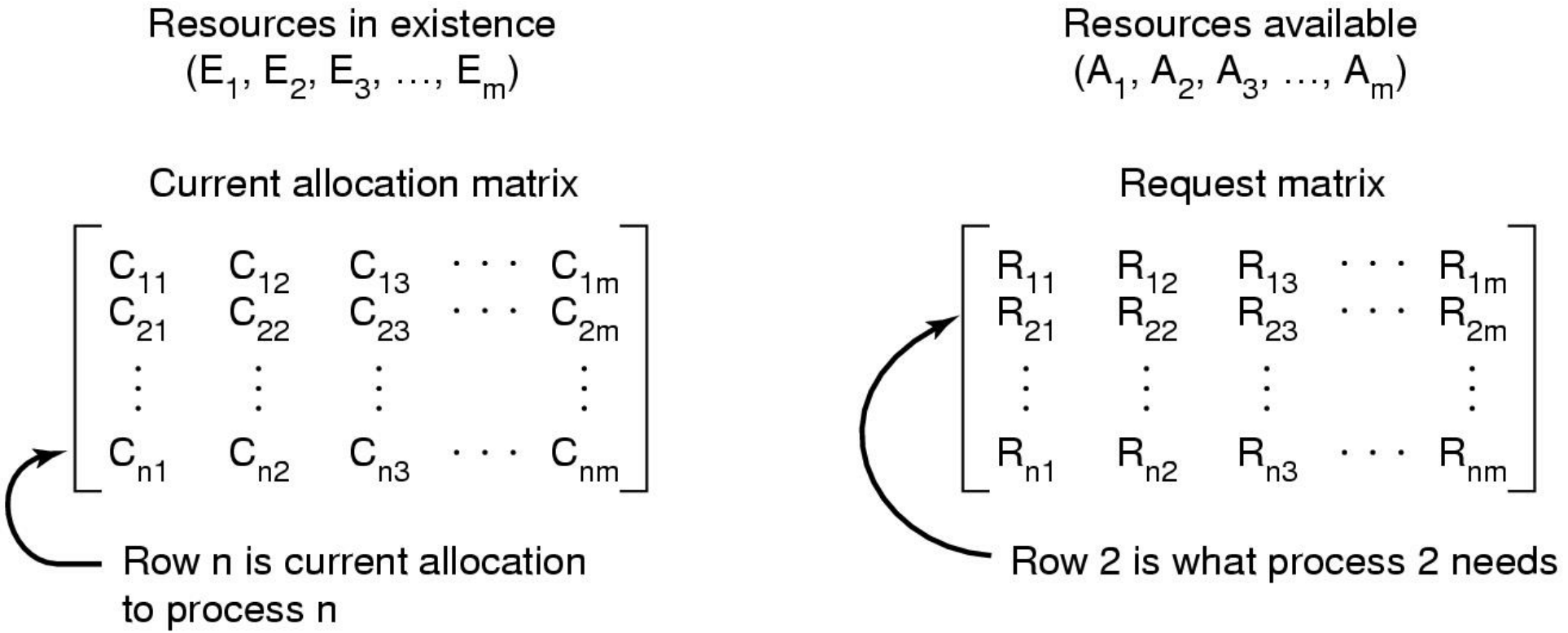
- Reasonable if
  - ① deadlocks occur very rarely
  - ② cost of prevention is high

Detection with One Resource of Each Type



- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock
- Many algorithms for detecting cycles in directed graphs.

Detection with Multiple Resources of Each Type



Data structures needed by deadlock detection algorithm

$$\Sigma C_{ij} + A_j = E_j$$

Deadlock Detection Algorithm

1. Recovery look for an unmark process P_i s.t. the i-th row of $\boldsymbol{R}$ is less than $\boldsymbol{A}$. (P_i 's request can be satisfied)
2. If such a process is found, add the i-th row of $\boldsymbol{C}$ to A, mark the process and go back to step 1. (When Pi completes, its resources will become available).
3. If no such process exist, the algorithm terminates. The unmarked process, if any, are deadlock.

Deadlock Detection Algorithm

```
Add all nodes to Unfinished;
Do{
    done = true
    foreach node in Unfinished{
        if([R_node] <= [A]){
            remove node from Unifished;
            [A] = [A] + [C_node]
            done = false;
        }
    }
}until(done)
```

Current Request Resources of node

Available Resources

Resources held by node