Lecture 11

Semaphores, Deadlocks & Memory Management

Semaphores · Producer-Consumer · Deadlock · Coffman · Banker's · Paging

SENG 21213 · Computer Architecture & Operating Systems

Learning Objectives — By the end of this lecture you should be able to:

1. Implement semaphore operations (semWait/semSignal) and explain the count semantics
2. Distinguish mutex from binary semaphore by ownership rules and use cases
3. Write the three-semaphore producer-consumer solution and explain the critical ordering rule
4. State all four Coffman conditions and describe a prevention strategy and trade-off for each
5. Apply the Banker's Algorithm safety check and request-grant procedure to a given state
6. Trace FIFO, LRU, and OPT page replacement for a reference string and count page faults

Section 1 · Semaphores and the Producer-Consumer Problem

📖 Stallings Reference

- Chapter 5: Concurrency — Mutual Exclusion and Synchronisation
- Section 5.3 — Semaphores
- Section 5.4 — Monitors (for comparison)

◆ Semaphore — Definition

- A semaphore is an integer variable s with two atomic operations: semWait(s) and semSignal(s).
- The key property: blocking (not spinning) — a process that cannot proceed is removed from the CPU and placed in a waiting queue. The CPU is free to run other processes.
- Counting semaphore: s can be any non-negative integer. Used for managing N identical resources.
- Binary semaphore: s ∈ {0,1}. Equivalent to a mutex lock for mutual exclusion.
- Strong semaphore: FIFO waiting queue — guarantees no starvation.
- Weak semaphore: arbitrary order — starvation possible.

1.1 Semaphore Operations

```
struct semaphore {
    int       count;    /* >0: resources available; <0: |count| processes waiting */
    queue_t   waiting;  /* queue of blocked process PCBs */
};

/* semWait (P / down / wait): try to acquire resource */
void semWait(semaphore *s) {
    s->count--;                    /* decrement (may go negative) */
    if (s->count < 0) {            /* no resource available */
        add_to_queue(s->waiting, current_process);
        block();                   /* remove from ready queue; context switch */
    }
}

/* semSignal (V / up / signal): release resource, wake a waiter */
void semSignal(semaphore *s) {
    s->count++;
    if (s->count <= 0) {           /* at least one process is waiting */
        pcb_t *p = remove_from_queue(s->waiting);
        wakeup(p);                 /* move p to ready queue */
    }
}
/* CRITICAL: semWait and semSignal must each be ATOMIC */
/* Implemented with interrupts disabled (uniprocessor) or test-and-set (SMP) */
```

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
         /* do nothing */;
     /* critical section */;
     bolt = 0;
     /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));
}
```

(a) Compare and swap instruction

```
/* program mutualexclusion */
int const n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
      int keyi = 1;
      do exchange (&keyi, &bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

Figure: Semaphore operation trace: counting semaphore tracking 3 processes and the wait queue dynamics (Stallings Fig. 5.11)

1.2 Mutex vs. Binary Semaphore

Property	Mutex	Binary Semaphore
Ownership	ONLY the thread that locked it can unlock it. Ownership is enforced.	Any thread can signal, regardless of which thread waited.

Property	Mutex	Binary Semaphore
Use case	Protecting a critical section that is entered and exited by the SAME thread.	Signalling between threads: consumer waits; producer signals.
Recursive locking	Recursive mutex: same thread can lock multiple times (counts locks). Deadlock if non-recursive mutex is locked twice by same thread.	Not recursive — second semWait by same thread will block if count=0.
Priority inheritance	Supported — OS can temporarily boost priority of mutex-holding thread to prevent priority inversion.	Not typically supported.
Example	pthread_mutex_t in pthreads; OS critical sections	sem_t in POSIX; Dijkstra's original semaphore

1.3 The Producer-Consumer Problem with Bounded Buffer

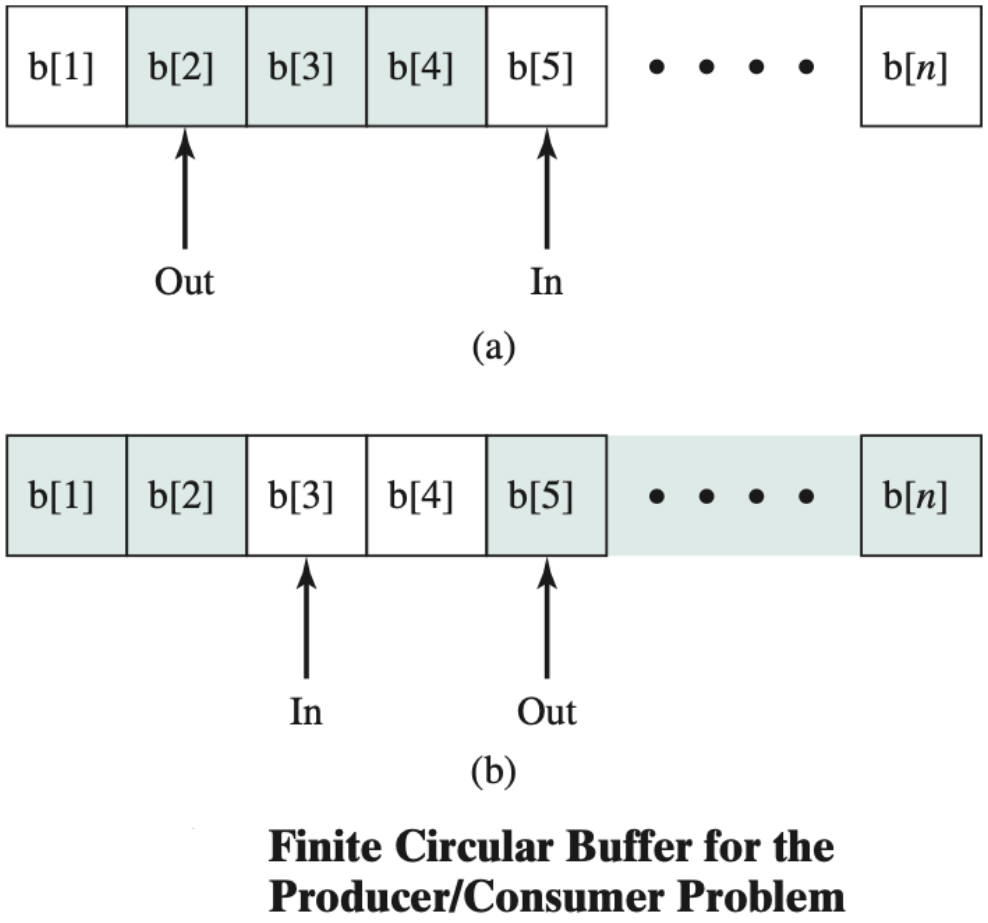

Figure: Bounded-buffer producer-consumer problem — shared circular buffer with head and tail pointers (Stallings Fig. 5.12)

```
/* Three semaphores: n (item count), e (empty slots), s (mutual exclusion) */
semaphore n = 0;            /* items currently in buffer (initially 0) */
semaphore s = 1;            /* binary mutex on buffer access (initially 1) */
semaphore e = BUFFER_MAX;   /* empty slots available (initially BUFFER_MAX) */

void producer() {
    while (true) {
        T = produce();
        semWait(e);    /* wait for an empty slot — blocks if buffer full */
        semWait(s);    /* LOCK: enter critical section */
        buffer[in] = T;
        in = (in + 1) % BUFFER_MAX;
        semSignal(s);  /* UNLOCK: leave critical section */
        semSignal(n);  /* signal: one more item available */
    }
}

void consumer() {
    while (true) {
        semWait(n);    /* wait for an item — blocks if buffer empty */
        semWait(s);    /* LOCK: enter critical section */
        T = buffer[out];
        out = (out + 1) % BUFFER_MAX;
        semSignal(s);  /* UNLOCK: leave critical section */
        semSignal(e);  /* signal: one more empty slot */
        consume(T);
    }
}
```

⚠ Critical Ordering in Producer-Consumer

- The order of semWait operations MATTERS. The producer MUST semWait(e) BEFORE semWait(s).
- If reversed (semWait(s) first, then semWait(e)): Producer holds mutex s, finds buffer full, blocks on e. Consumer cannot acquire s to remove an item. DEADLOCK.
- Rule: Always acquire the counting semaphore BEFORE the mutex semaphore.

```
struct semaphore {
       int count;
       queueType queue;
};
void semWait(semaphore s)
{
       s.count--;
       if (s.count < 0) {
         /* place this  process in s.queue */;
         /* block this  process */;
       }
}
void semSignal(semaphore s)
{
       s.count++;
       if (s.count<= 0) {
         /* remove a process P from s.queue */;
         /* place process P on ready list */;
       }
}
```

Figure: Mutual exclusion using semaphores — three concurrent processes accessing a shared critical section (Stallings Fig. 5.15)

Section 2 · Deadlocks

📖 Stallings Reference

- Chapter 6: Concurrency — Deadlock and Starvation
- Section 6.1 — Principles of Deadlock
- Section 6.4 — Deadlock Detection

◆ Deadlock — Definition

- A DEADLOCK exists when EVERY process in a set S is blocked waiting for an event that can ONLY be caused by another process in the same set S.
- No process in S can ever proceed — the system is permanently stuck.
- Starvation differs: A single process waits indefinitely for a resource while OTHER processes outside the deadlock set continue — the stuck process is merely neglected, not causing all others to wait.

2.1 Joint Progress Diagram

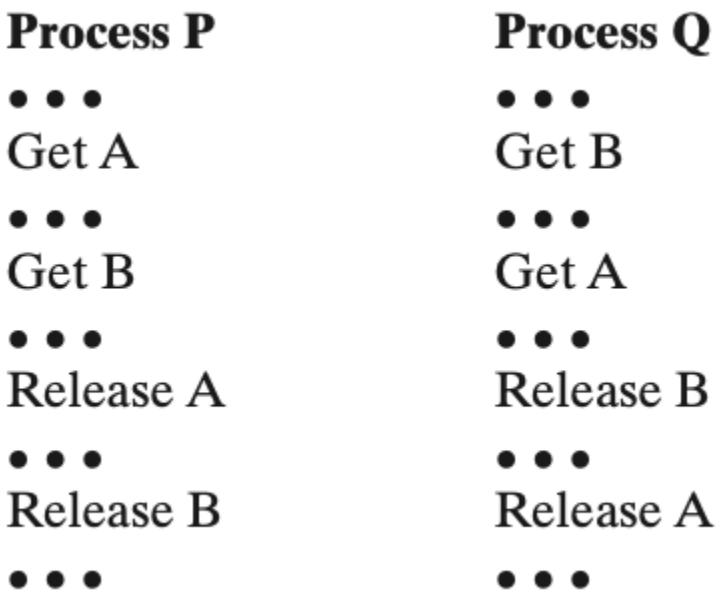

Figure: Joint progress diagram: two processes P and Q competing for resources Ra and Rb. The fatal region is the deadlock zone — neither process can proceed once inside (Stallings Fig. 6.2)

The diagram shows two axes (P's progress and Q's progress). The fatal region represents system states where P holds Ra and needs Rb, while Q holds Rb and needs Ra — circular wait. Of the six possible execution paths, only those that avoid the fatal region are deadlock-free.

2.2 Reusable vs. Consumable Resources

Resource Type	Definition	Examples	Deadlock Scenario
Reusable	Used by one process at a time; not depleted — returned after use.	CPU, memory, disk drives, printers, database locks, mutex locks	P holds disk, needs tape. Q holds tape, needs disk. Neither releases — deadlock.
Consumable	Created by one process and consumed (destroyed) by another.	Interrupts, signals, messages, data in I/O buffers	P waits for message from Q. Q waits for message from P. Neither sends first — deadlock.

Figure: Resource allocation graph examples showing potential deadlock situations with reusable and consumable resources (Stallings Fig. 6.3)

2.3 The Four Coffman Conditions

All four conditions must hold SIMULTANEOUSLY for deadlock to occur. Preventing any one prevents deadlock:

Condition	Description	Prevention Strategy	Trade-off
1. Mutual Exclusion	At least one resource held non-shareably — only one process at a time.	Make resources shareable where possible (read-only files; print spooler).	Cannot always share — printers cannot be used simultaneously.
2. Hold and Wait	A process holds ≥1 resource while waiting to acquire more.	All-or-nothing: request all resources before starting. OR: release all held resources before requesting new ones.	Low resource utilisation (holds resources it may not use for hours). Starvation if large resource set hard to acquire simultaneously.
3. No Preemption	Resources cannot be forcibly taken from a process — only voluntarily released.	OS may preempt resources from blocked processes (save state and restore when resources available again).	Only works for preemptable resources (CPU, memory). Cannot preempt printers mid-job.
4. Circular Wait	Circular chain: P1 waits for P2, P2 waits for P3, …, Pn waits for P1.	Impose total ordering on resource types. All processes request resources in increasing order number.	Some processes must request resources in unnatural order, reducing modularity.

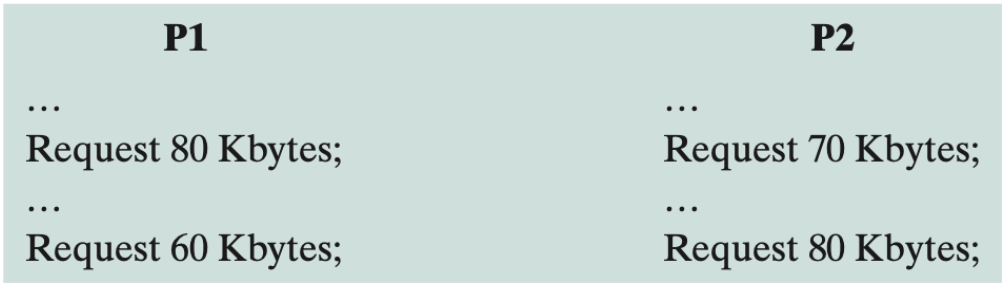

Figure: Safe state (all processes can complete) vs. unsafe state (deadlock possible but not guaranteed) for Banker's Algorithm (Stallings Fig. 6.7)

2.4 Deadlock Avoidance — The Banker's Algorithm

◆ Safe State

- A state is SAFE if there exists at least one execution sequence (a 'safe sequence') in which every process can obtain its maximum resource needs, run to completion, and release its resources — without deadlock.
- The Banker's Algorithm: before granting a resource request, simulate the allocation. If the

resulting state is safe → grant. If unsafe → defer (process must wait).

- Key data: Allocation matrix (what each process currently holds), Max matrix (maximum it will ever need), Available vector (unallocated resources).
- Need matrix = Max − Allocation.

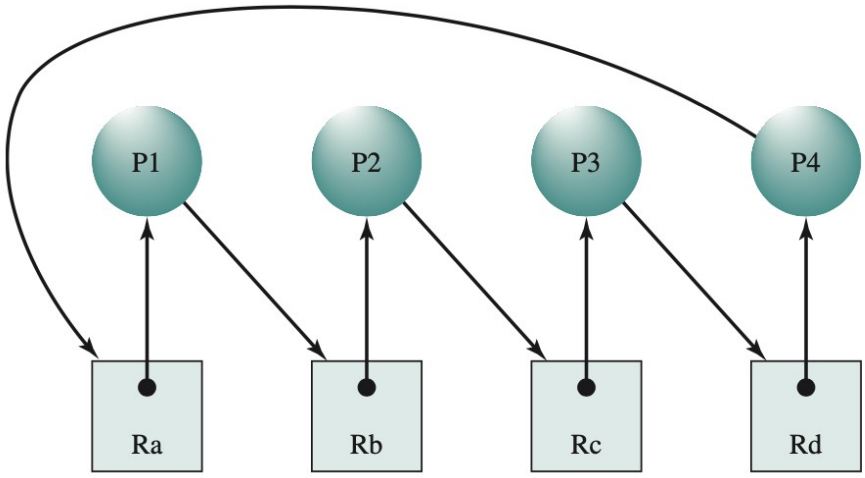

Figure: Banker's Algorithm example showing safe state determination for 5 processes and 3 resource types (Stallings Fig. 6.8)

2.5 Starvation

Starvation is not deadlock — the system makes progress, but one unfortunate process never gets scheduled.

Property	Deadlock	Starvation
Progress	NO process in the deadlock set makes progress	Other processes make progress; victim does not
Scope	Set of mutually waiting processes	Single neglected process
Fix	Prevention, avoidance, or detection+recovery	Ageing: raise priority of waiting processes over time
Recovery	OS must kill a process or preempt a resource	Simply give the starved process a higher priority

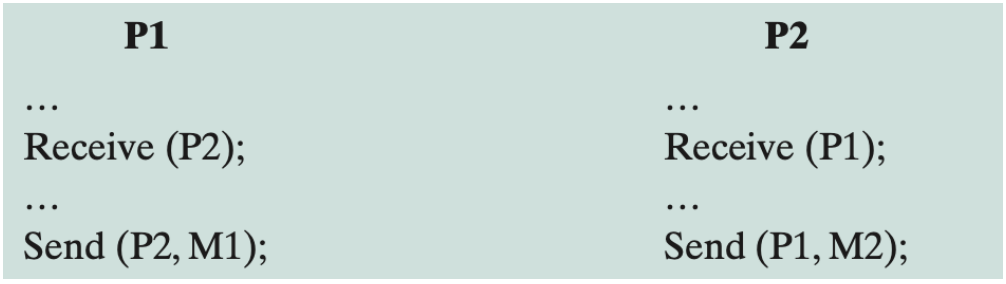

Figure: Starvation scenario: P2 indefinitely bypassed as P1 and P3 alternate access to the shared resource

Section 3 · Memory Management Fundamentals

3.1 OS Memory Management Responsibilities

1. Process Isolation: prevent one process reading/writing another's memory. Hardware enforced via page table permission bits.
2. Automatic Allocation: allocate and deallocate memory dynamically as processes are created and destroyed. No manual memory management by processes.
3. Modular Programming Support: allow dynamic linking, shared libraries, stack growth on demand.
4. Protection and Access Control: kernel memory not accessible from user mode; read-only code segments cannot be written; stack not executable (NX/XD bit).
5. Long-Term Storage: file system provides persistent storage surviving process termination and power cycles.

3.2 Paging

Concept	Description
Page / Frame	Physical memory divided into fixed-size frames (typically 4 KB). Process virtual address space divided into equal-size pages.
Page Table	One entry per virtual page. Maps page number → frame number. Holds permission bits (R/W/X, user/kernel, present/absent).
TLB (Translation Lookaside Buffer)	Hardware cache of recent page table entries. Eliminates memory access for page table lookup on hits. 99%+ TLB hit rate typical.
Page Fault	CPU raises page fault exception when accessing a page not in memory (present bit=0). OS loads page from disk (swapping), updates page table, retries instruction.
Virtual Address Split (32-bit)	4 KB pages: [page number — 20 bits | byte offset — 12 bits]. EA = frame_base + offset.

3.3 Page Replacement Algorithms

Algorithm	Strategy	Belady's Anomaly?	Performance
OPT (Optimal)	Replace page not needed for the longest future time.	No	Best possible — requires future knowledge; used as benchmark only
FIFO	Replace the page that has been in memory the longest.	YES	Simple but poor — may evict a heavily-used old page
LRU	Replace the page that was used least recently.	No	Near-optimal for most workloads; expensive to implement exactly
Clock (Second Chance)	Circular list + reference bit R. Replace first page with R=0; set R=0 as clock sweeps past.	No	Good approximation of LRU; practical; used in Linux (active/inactive lists)
LFU	Replace the page used least frequently.	No	Keeps heavily-used pages; can keep dead pages that were

Algorithm	Strategy	Belady's Anomaly?	Performance
			popular early

Section 4 · OS Assignment — Stage 3

★ Milestone 3: Physical Memory Manager (PMM)

- pmm.h — Interface: void pmm_init(uint32_t memsize); uint32_t pmm_alloc_frame(); void pmm_free_frame(uint32_t addr);
- pmm.c — Bitmap implementation: one bit per 4 KB physical frame. Bit=1 means used; bit=0 means free.
- pmm_init(): mark frames 0 through KERNEL_END/4096 as used (kernel code, stack, PCBs).
- pmm_alloc_frame(): scan bitmap for first 0 bit; set it to 1; return frame_number * 4096.
- pmm_free_frame(addr): clear bit at addr/4096.
- Integration: In create_process(), call pmm_alloc_frame() to allocate each process's 4 KB stack.
- Test: Allocate 10 frames, free frames 3 and 7, allocate 2 more — verify addresses 0xC000 and 0x1C000 are returned (reuse of freed frames).

Section 5 · Practice Exercises

✎ Exercise 11.1 [14 marks] — Semaphores

- (a) [4] Trace the following sequence of semaphore operations on a counting semaphore s (initially s=1): P1:semWait(s), P2:semWait(s), P3:semWait(s), P1:semSignal(s), P4:semWait(s), P2:semSignal(s). After each operation, record: s.count value; which processes are in the waiting queue; which processes are running.
- (b) [4] Demonstrate the deadlock that occurs if the producer-consumer solution is implemented with the semWait operations in WRONG ORDER (s acquired before e): (i) Start: s=1, n=0, e=0 (full buffer). Show the sequence of operations and explain why neither producer nor consumer can proceed. (ii) What is the correct order and why does it prevent deadlock?
- (c) [6] The Readers-Writers problem: multiple readers may simultaneously read a shared database, but a writer needs exclusive access. Define the semaphores needed and write pseudocode for reader() and writer() processes. Your solution must: (i) allow concurrent reads, (ii) ensure write exclusivity, (iii) give writers priority over waiting readers to prevent writer starvation.

✎ Exercise 11.2 [12 marks] — Deadlock

- (a) [4] A system has 4 processes (P1–P4) and 3 resource types: R1(3 units), R2(2 units), R3(2 units). Current allocation and maximum need: P1: allocated(1,0,0), max(3,2,2); P2: allocated(0,1,0), max(0,2,0); P3: allocated(1,1,0), max(1,1,2); P4: allocated(0,0,1), max(0,0,2). Available: (1,0,1). Apply the Banker's Algorithm safety check: is the state safe? If yes, give the safe sequence. If no, explain.
- (b) [4] P2 now requests (0,1,0). (i) Can the request be granted immediately without checking? (ii) Apply the Banker's Algorithm: simulate granting the request and check if the resulting state is

safe. (iii) Should the request be granted? Justify.

- (c) [4] Explain the practical limitations of the Banker's Algorithm that prevent its use in real-world OS. For each of the four Coffman conditions, describe a DIFFERENT real-world OS technique (e.g. Linux, Windows, database system) that prevents or avoids that specific condition.

✎ Exercise 11.3 [9 marks] — Paging and Page Replacement

- (a) [4] A page table for a process with 32-bit virtual addresses and 4 KB pages is checked for the address 0x12345ABC. (i) Split the address into page number and byte offset. (ii) Page table entry at that page number is: frame=0x00567, R=1, W=1, X=0, Present=1. What is the physical address? (iii) If a write occurs, is it permitted? (iv) If Present=0, what happens?
- (b) [5] A process references pages in this order: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Physical memory holds 3 frames. (i) Apply FIFO and count page faults. (ii) Apply LRU and count page faults. (iii) Apply OPT and count page faults. Show the frame state after each reference. (iv) Does FIFO suffer Belady's anomaly with 4 frames? Verify by repeating the FIFO simulation.