COT 4600 Operating Systems Fall 2009

Dan C. Marinescu

Office: HEC 439 B

Office hours: Tu-Th 3:00-4:00 PM

Lecture 17

Reading assignments: Chapter 5.1, 5.2 and 5.3

Phase 2 of the project and HW4 due on Thursday, October, 22

- Last time:
  - Midterm
- Today:
  - Virtualization
    - Threads
    - Virtual memory
    - Bounded buffers
  - Virtual Links
  - Thread coordination with a bounded buffer
  - Race conditions
- Next Time:
  - Enforcing Modularity in Memory

Virtualization – relating physical with virtual objects

- Virtualization → simulating the interface to a physical object by:
  1. Multiplexing → create multiple physical objects from one instance of a physical object.
  2. Aggregation → create one virtual object from multiple physical objects
  3. Emulation → construct a virtual object from a different type of a physical object. Emulation in software is slow.

Method	Physical Resource	Virtual Resource
Multiplexing	processor	thread
	real memory	virtual memory
	communication channel	virtual circuit
	processor	server (e.g., Web server)
Aggregation	disk	RAID
	core	multi-core processor
Emulation	disk	RAM disk
	system (e.g. Macintosh)	virtual machine (e.g., Virtual PC)
Multiplexing + Emulation	real memory + disk	virtual memory with paging
	communication channel + processor	TCP protocol

Virtualization of the three abstractions. (1) Threads

- Implemented by the operating system for the three abstractions:

1. Threads → a thread is a virtual processor; a module in execution
    1. Multiplexes a physical processor
    2. The state of a thread: (1) the reference to the next computational step (the Pc register) + (2) the environment (registers, stack, heap, current objects).
    3. Sequence of operations:
        1. Load the module's text
        2. Create a thread and lunch the execution of the module in that thread.
    4. A module may have several threads.
    5. The thread manager implements the thread abstraction.
        1. Interrupts → processed by the interrupt handler which interacts with the thread manager
        2. Exception → interrupts caused by the running thread and processed by exception handlers
        3. Interrupt handlers run in the context of the OS while exception handlers run in the context of interrupted thread.

Virtualization of the three abstractions. (2) Virtual memory

2. Virtual Memory → a scheme to allow each thread to access only its own virtual address space (collection of virtual addresses).
    1. Why needed:
        1. To implement a memory enforcement mechanism; to prevent a thread running the code of one module from overwriting the data of another module
        2. The physical memory may be too small to fit an application; otherwise each application would need to manage its own memory.
    2. Virtual memory manager → maps virtual address space of a thread to physical memory.

    Thread + virtual memory allow us to create a virtual computer for each module.

    Each module runs in own address space; if one module runs mutiple threads all share one address space.

Virtualization of the three abstractions: (3) Bounded buffers

3. Bounded buffers → implement the communication channel abstraction
   1. Bounded→ the buffer has a finite size. We assume that all messages are of the same size and each can fit into a buffer cell. A bounded buffer will only accommodate N messages.
   2. Threads use the SEND and RECEIVE primitives.

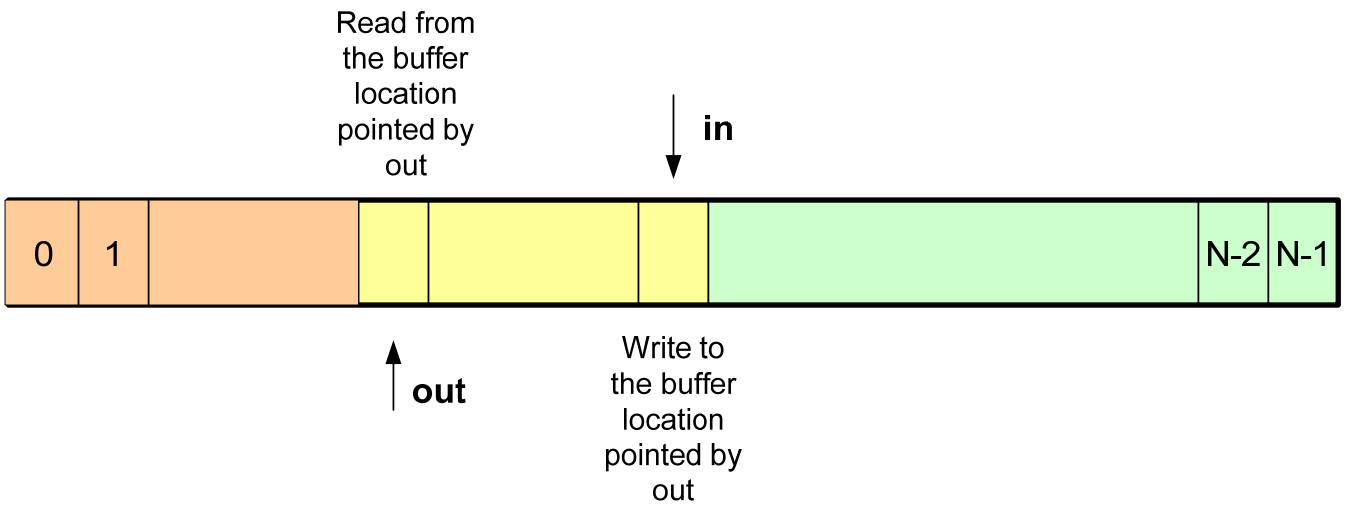
Principle of least astonishment

- Study and understand simple phenomena or facts before moving to complex ones. For example:
  - Concurrency → an application requires multiple threads that run at the same time. Tricky. Understand sequential processing first.
  - Examine a simple operating system interface to the three abstractions

Memory	CREATE/DELETE_ADDRESS SPACE ALLOCATE/FREE_BLOCK MAP/UNMAP UNMAP
Interpreter	ALLOCATE_THREAD DESTROY_THREAD EXIT_THREAD YIELD AWAIT ADVANCE TICKET ACQUIRE RELEASE
Communication channel	ALLOCATE/DEALLOCATE_BOUNDED_BUFFER SEND/RECEIVE

Thread coordination with a bounded buffer

- Producer-consumer problem → coordinate the sending and receiving threads
- Basic assumptions:
  - We have only two threads
  - Threads proceed concurrently at independent speeds/rates
  - Bounded buffer – only N buffer cells
  - Messages are of fixed size and occupy only one buffer cell
- Spin lock → a thread keeps checking a control variable/semaphore "until the light turns green"




```
shared structure buffer
    message instance message[N]
    integer in initially 0
    integer out initially 0

procedure SEND (buffer reference p, message instance msg)
    while p.in – p.out = N do nothing      /* if buffer full wait
    p.message [p.in modulo N] ←msg         /* insert message into buffer cell
    p.in  ← p.in + 1                       /* increment pointer to next free cell

procedure RECEIVE (buffer reference p)
    while p.in = p.out   do nothing        /* if buffer empty wait for message
    msg← p.message [p.in modulo N]         /* copy message from buffer cell
    p.out ← p.out + 1                      /* increment pointer to next message
    return msg
```

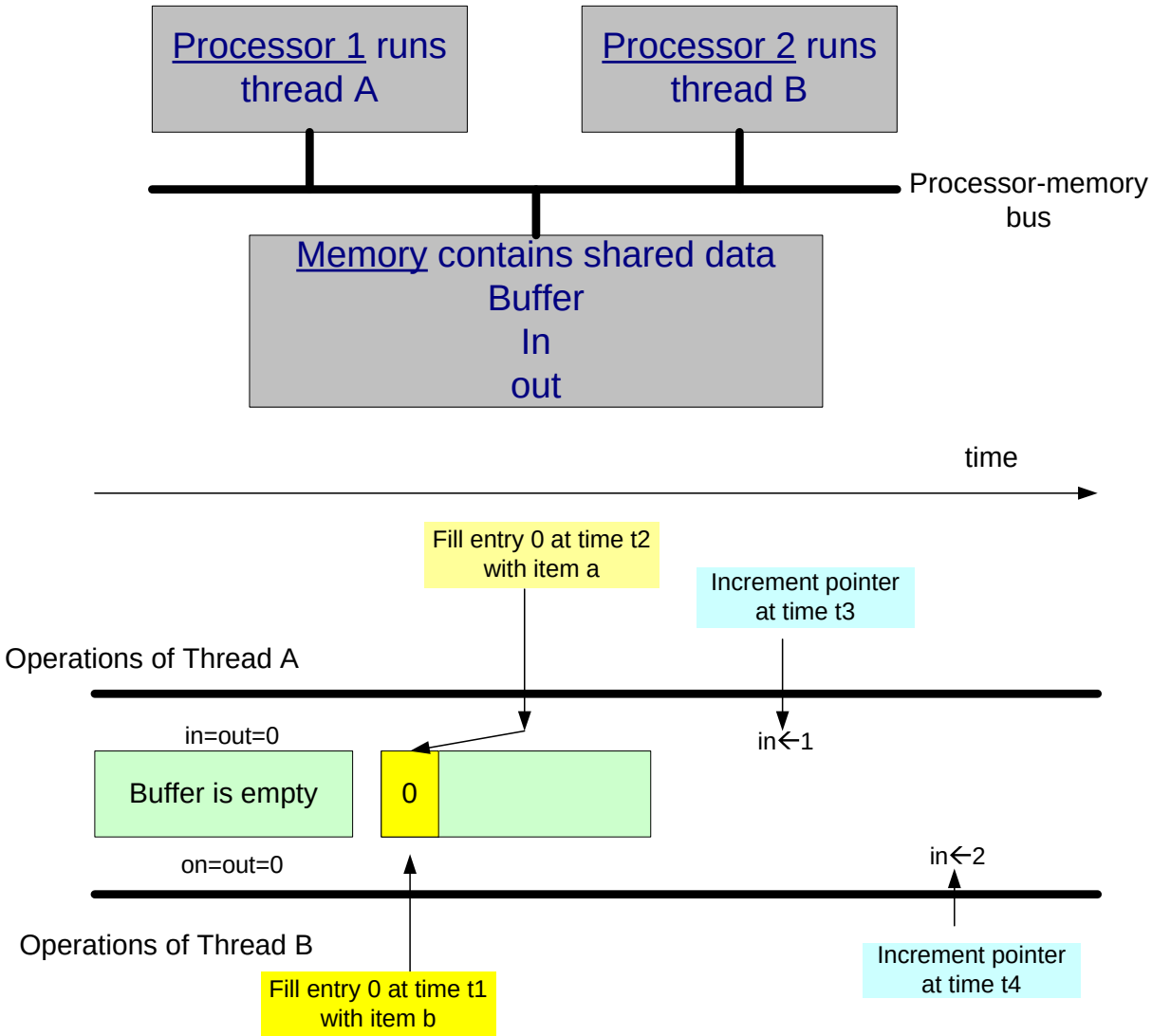
Implicit assumptions for the correctness of the implementation

1. One sending and one receiving thread. Only one thread updates each shared variable.
2. Sender and receiver threads run on different processors→ to allow spin locks
3. **in** and **out** are implemented as integers large enough so that they do not overflow (e.g., 64 bit integers)
4. The shared memory used for the buffer provides read/write coherence
5. The memory provides before-or-after atomicity for the shared variables **in** and **out**
6. The result of executing a statement becomes visible to all threads in program order. No compiler optimization supported

Race conditions

- Race condition → error that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly.
- Race conditions depend on the exact timing of events thus are not reproducible.
  - A slight variation of the timing could either remove a race condition or create.
  - Very hard to debug such errors.

Two senders execute the code concurrently

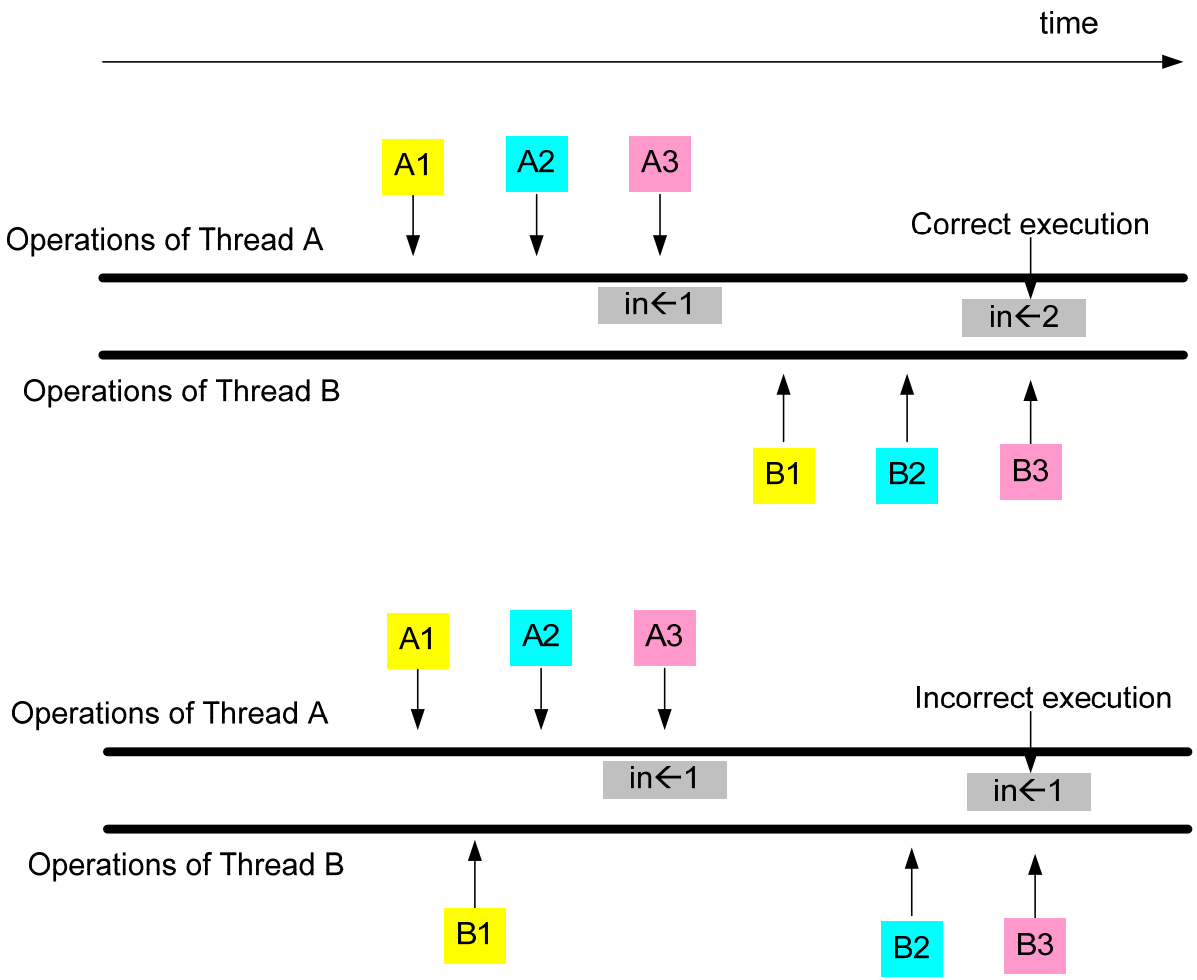


Item b is overwritten, it is lost

Another manifestation of race conditions
→ incrementing a pointer is not atomic

in ← in+1

```
1.  L    R1,in
2.  ADD  R1,1
3.  ST   R1,in
```



One more pitfall of the previous implementation of bounded buffer

- If **in** and **out** are long integers (64 or 128 bit) then a load requires two registers, e.,g, R1 and R2.

  int “00000000FFFFFFFF”

  L R1,int /* R1 ← 00000000

  L R2,int+1 /* R2 ← FFFFFFFF

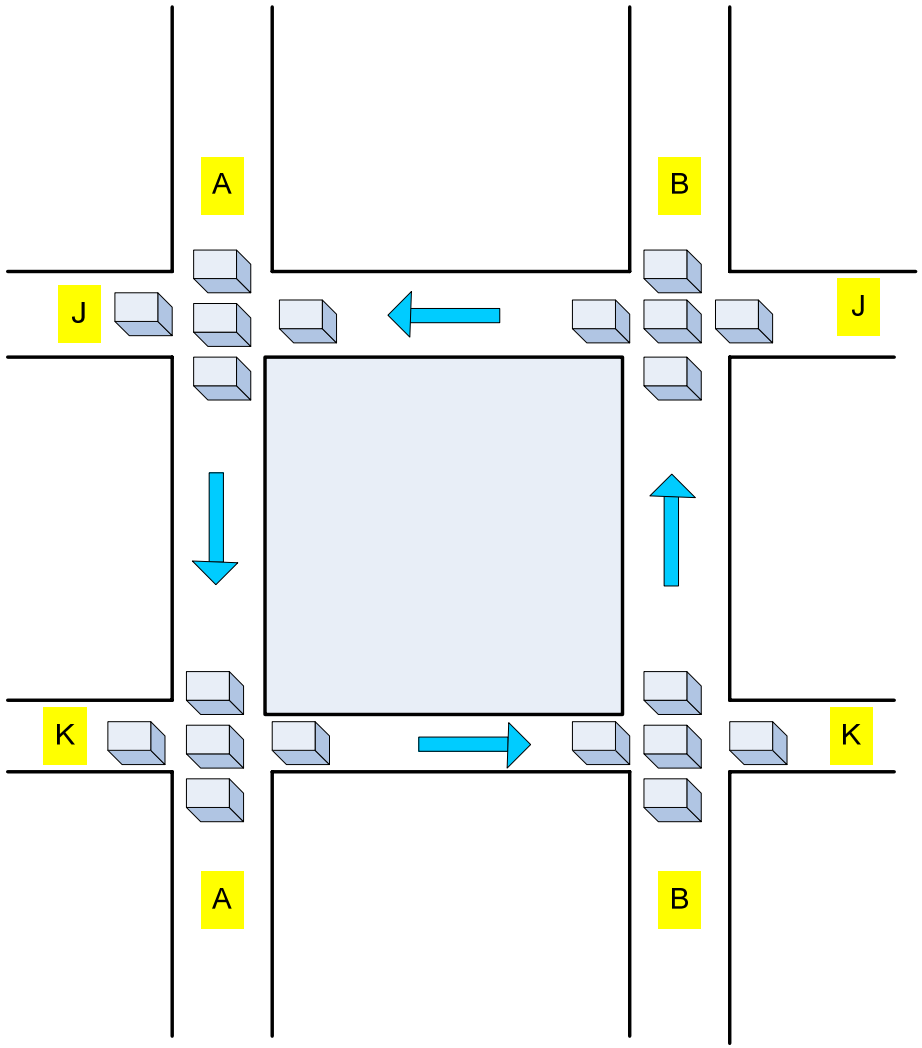
- Race conditions could affect a load or a store of the long integer.

Lock

- Lock → a mechanism to guarantee that a program works correctly when multiple threads execute concurrently
  - a multi-step operation protected by a lock behaves like a single operation
  - can be used to implement before-or after atomicity
  - shared variable acting as a flag (traffic light) to coordinate access to a shared variable
  - works only if all threads follow the rule → check the lock before accesing a shared variable.

Deadlocks

- Happen quite often in real life and the proposed solutions are not always logical: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.” a pearl from Kansas legislation.
- Deadlock jury.
- Deadlock legislative body.


A
B
J
J
K
K
A
B

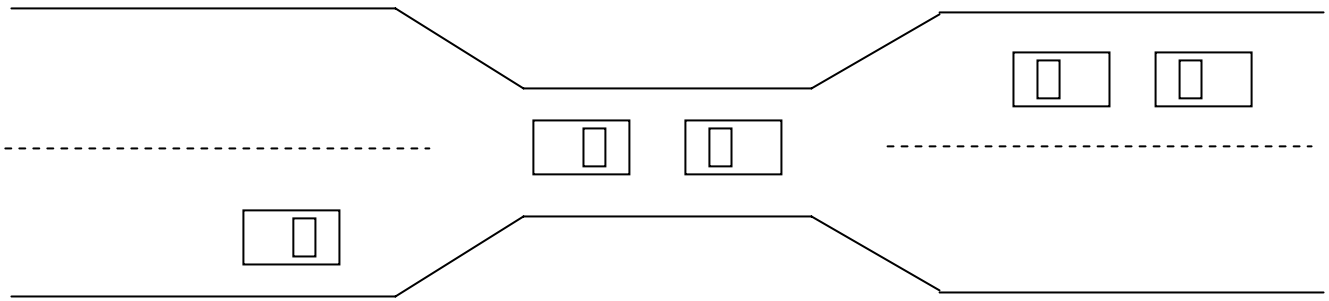
Deadlocks

- Deadlocks ➔ prevent sets of concurrent threads/processes from completing their tasks.
- How does a deadlock occur ➔ a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

- Aim➔ prevent or avoid deadlocks

Example of a deadlock



- Traffic only in one direction.
- Solution ➔ one car backs up (preempt resources and rollback). Several cars may have to be backed up .
- Starvation is possible.

System model

- Resource types R_1, R_2, . . ., R_m *(CPU cycles, memory space, I/O devices)*
- Each resource type R_i has W_i instances.
- Resource access model:
  - request
  - use
  - release

Simultaneous conditions for deadlock

- 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 (presumably after that process has finished).
- Circular wait**:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.