ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

OPERATING SYSTEMS & PARALLEL COMPUTING

Cooperating Processes

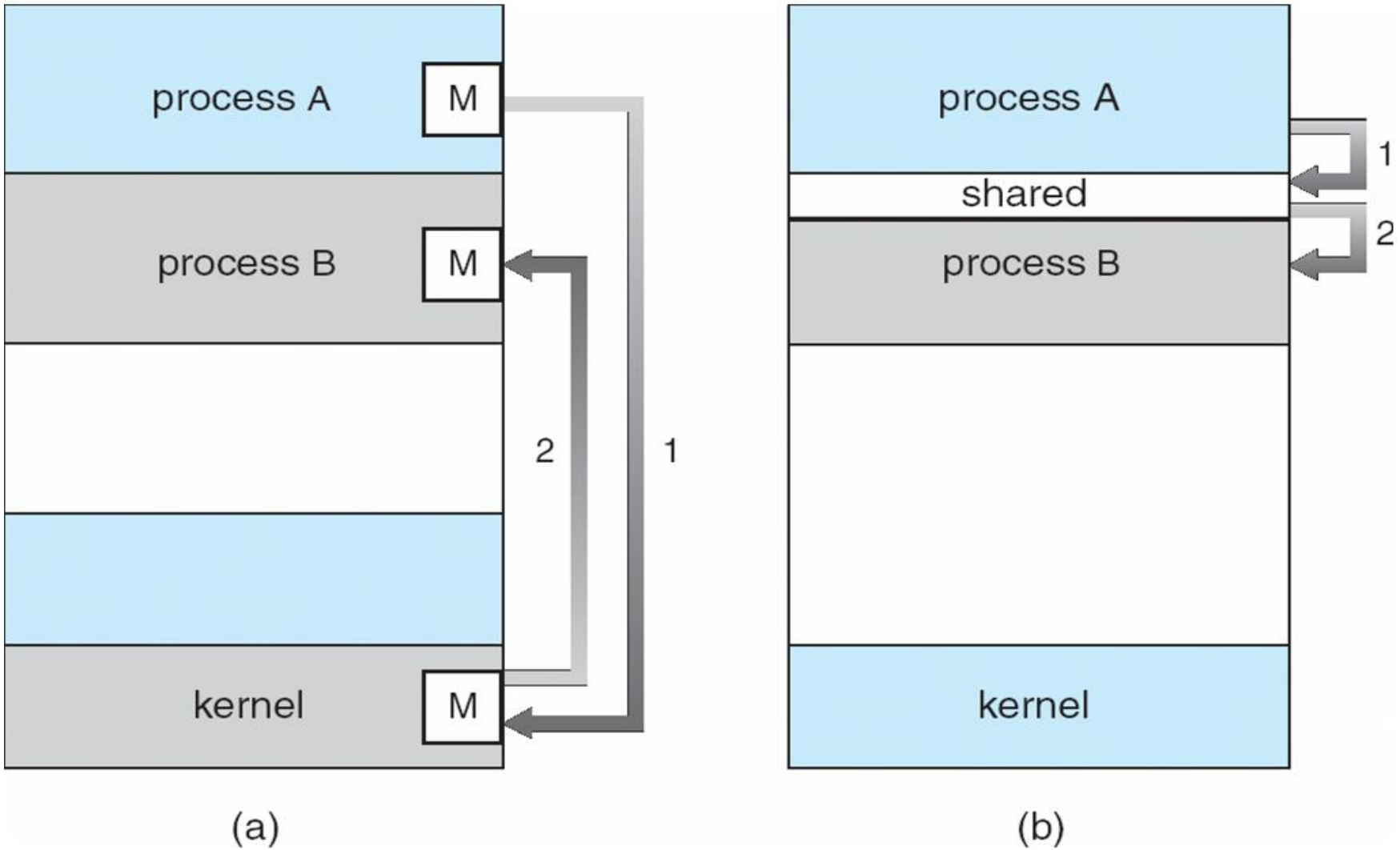
ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Introduction to Cooperating Process

Introduction to Cooperating Processes

- Processes within a system may be independent or cooperating.
- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by other processes, including sharing data.
- Reasons for cooperating processes:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Cooperation Models



Cooperation among Processes by Sharing

- Processes use and update shared data such as shared variables, memory, files, and databases.
- Writing must be mutually exclusive to prevent a race condition leading to inconsistent data views.
- Critical sections are used to provide this data integrity.
- A process requiring the critical section must not be delayed indefinitely; no deadlock or starvation.

Cooperation among Processes by Communication

- Communication by messages provides a way to synchronize, or coordinate, the various activities.
- Possible to have deadlock –
  - each process waiting for a message from the other process.
- Possible to have starvation –
  - two processes sending a message to each other while another process waits for a message.

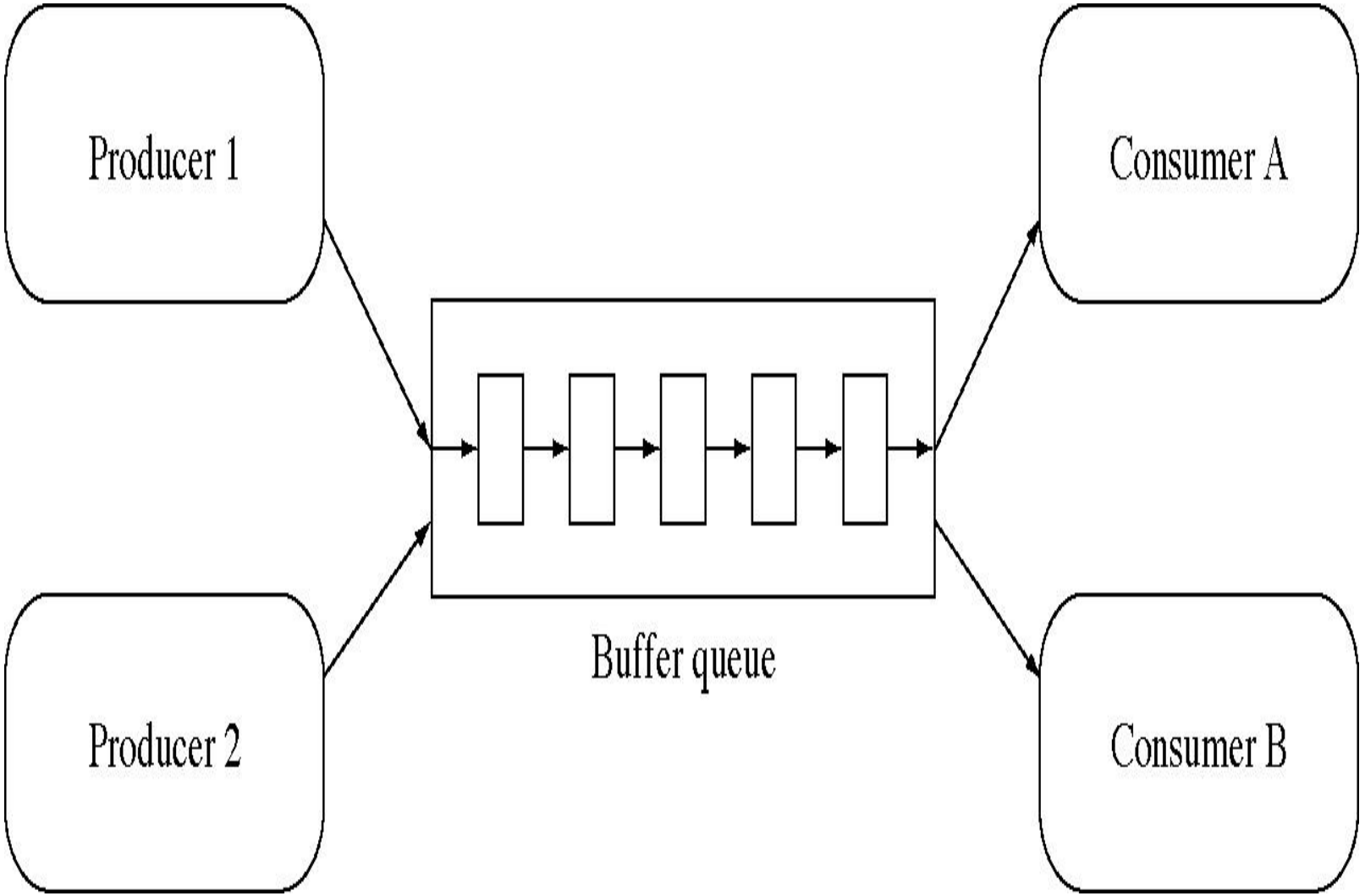
Producer/Consumer (P/C) Problem (1)

- Paradigm for cooperating processes – *Producer* process produces information that is consumed by a *Consumer* process.
  - Example 1: a print program produces characters that are consumed by a printer.
  - Example 2: an assembler produces object modules that are consumed by a loader.
  - Example 3: a client produces a message that a server could consume it.

Producer/Consumer (P/C) Problem (2)

- We need a buffer to hold items that are produced and later consumed:
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.

Multiple Producers and Consumers

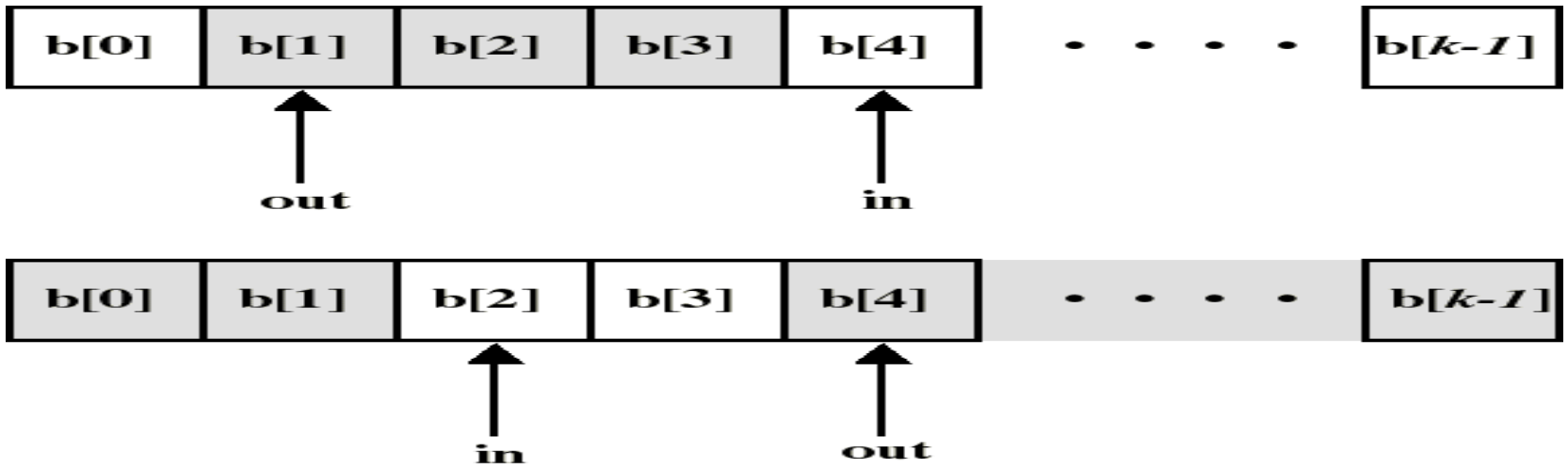


Producer/Consumer (P/C) Dynamics

- A producer process produces information that is consumed by a consumer process.
- At any time, a producer activity may create some data.
- At any time, a consumer activity may want to accept some data.
- The data should be saved in a buffer until they are needed.
- If the buffer is finite, we want a producer to block if its new data would overflow the buffer.
- We also want a consumer to block if there are no data available when it wants them.

Idea for Producer/Consumer Solution

- The bounded buffer is implemented as a circular array with 2 logical pointers: **`in`** and **`out`**.
- The variable **`in`** points to the next free position in the buffer.
- The variable **`out`** points to the first full position in the buffer.



Problems with concurrent execution

- Concurrent processes (or threads) often need to share data (maintained either in shared memory or files) and resources.
- If there is no controlled access to shared data, some processes will obtain an inconsistent view of this data.
- The action performed by concurrent processes will then depend on the order in which their execution is interleaved.

Example of inconsistent view

- 3 variables: A, B, C which are shared by thread T1 and thread T2.
- T1 computes C = A+B.
- T2 transfers amount X from A to B
  - T2 must do: A = A-X and B = B+X (so that A+B is unchanged).
- But if T1 computes A+B after T2 has done A = A-X but before B = B+X.
- Then T1 will not obtain the correct result for C = A+B.

Data Consistency

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffer. We can do so by having an integer count that keeps track of the number of items in the buffer.
- Initially, the count is set to 0. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes a item.

This is the Race Condition

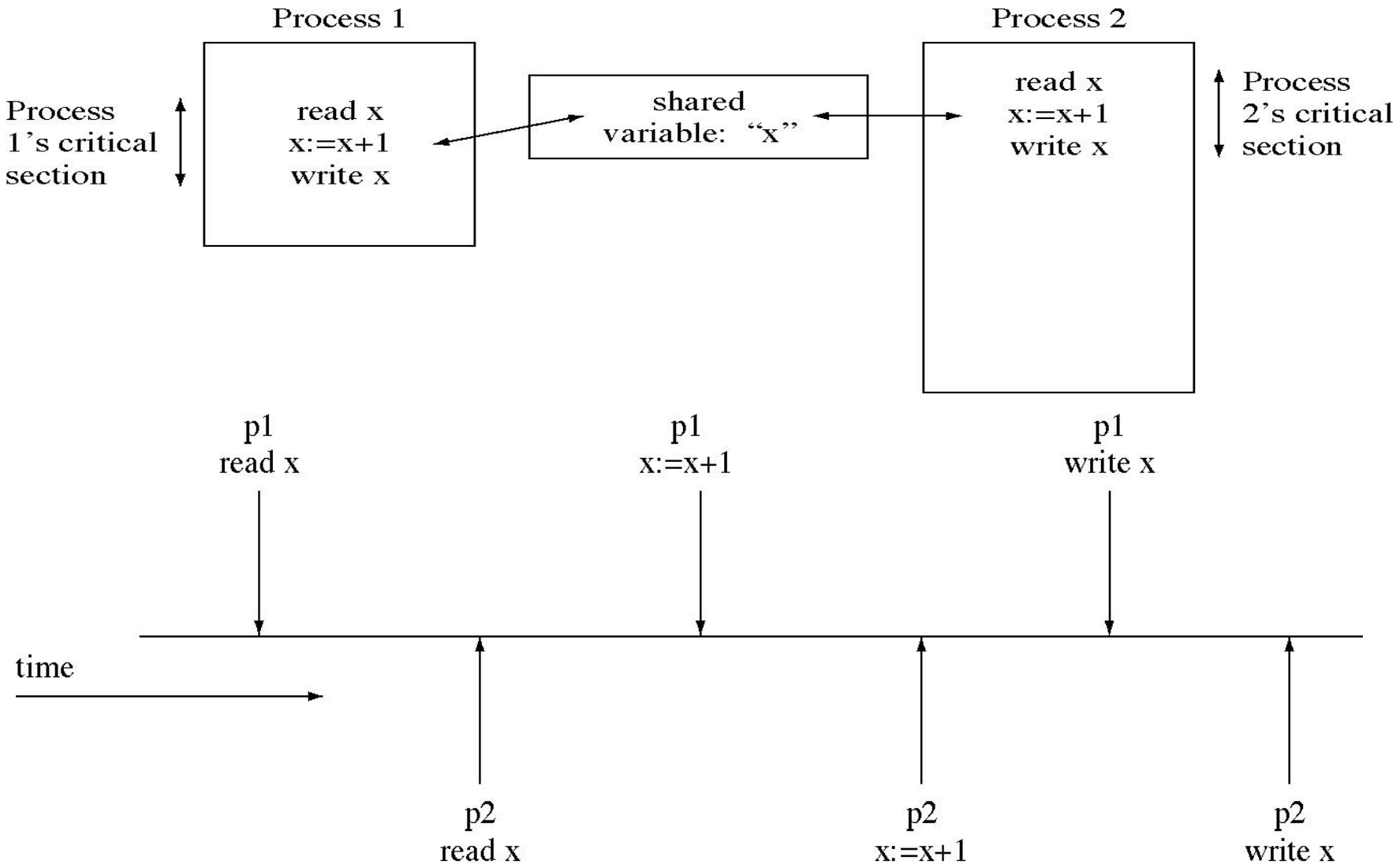
- **Race condition**: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must coordinate  or be **synchronized**.

Race condition updating a variable (1)

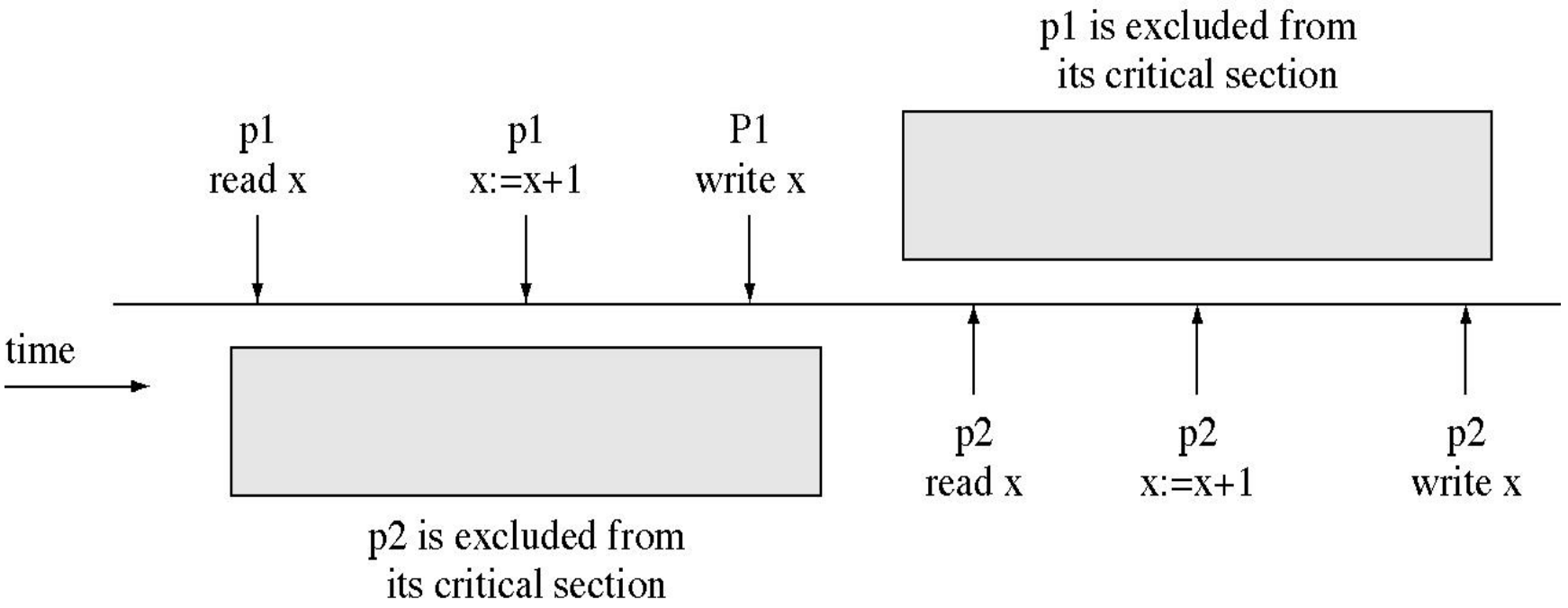
```
shared double balance;
Code for p1:
. . .
balance += amount;
. . .
Code for p1:
. . .
Load     R1, balance
Load     R2, amount
Add      R1, R2
Store    R1, balance
.   .   .
```

```
Code for p2:
. . .
balance += amount;
. . .
Code for p2:
. . .
Load     R1, balance
Load     R2, amount
Add      R1, R2
Store    R1, balance
.   .   .
```

Race condition updating a variable (2)



Critical section to prevent a race condition



- Multiprogramming allows logical parallelism, uses devices efficiently but we lose correctness when there is a race condition.
- So we forbid logical parallelism inside critical section so we lose some parallelism but we regain correctness.

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

The critical-Section Problem

The Critical-Section Problem

- *n* processes competing to use some shared data.
- No assumptions may be made about speeds or the number of CPUs.
- Each process has a code segment, called *Critical Section (CS)*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.

CS Problem Dynamics (1)

- When a process executes code that manipulates shared data (or resource), we say that the process is in it's Critical Section (for that shared data).
- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple processors).
- So each process must first request permission to enter its critical section.

CS Problem Dynamics (2)

- The section of code implementing this request is called the Entry Section (ES).
- The critical section (CS) might be followed by a Leave/Exit Section (LS).
- The remaining code is the Remainder Section (RS).
- The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors).

General structure of process P_i (other is P_j)

do {

entry section

critical section

leave section

remainder section

} **while (TRUE);**

- Processes may share some common variables to synchronize their actions.

Solution to Critical-Section Problem

- There are 3 requirements that must stand for a correct solution:
  1. **Mutual Exclusion**
  2. **Progress**
  3. **Bounded Waiting**
- We can check on all three requirements in each proposed solution, even though the non-existence of each one of them is enough for an incorrect solution.

Solution to CS Problem – Mutual Exclusion

1. **Mutual Exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

- Implications:
  - Critical sections better be focused and short.
  - Better not get into an infinite loop in there.
  - If a process somehow halts/waits in its critical section, it must not interfere with other processes.

Solution to CS Problem – Progress

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:
   - If only one process wants to enter, it should be able to.
   - If two or more want to enter, one of them should succeed.

Solution to CS Problem – Bounded Waiting

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.

Types of solutions to CS problem

- Software solutions –
  - algorithms who's correctness does not rely on any other assumptions.
- Hardware solutions –
  - rely on some special machine instructions.
- Operating System solutions –
  - provide some functions and data structures to the programmer through system/library calls.
- Programming Language solutions –
  - Linguistic constructs provided as part of a language.

What about process failures?

- If all 3 criteria (ME, progress, bounded waiting) are satisfied, then a valid solution will provide robustness against failure of a process in its remainder section (RS).
  - since failure in RS is just like having an infinitely long RS.
- However, no valid solution can provide robustness against a process failing in its critical section (CS).
  - A process Pi that fails in its CS does not signal that fact to other processes: for them Pi is still in its CS.

Drawbacks of software solutions

- Software solutions are very delicate ☺.
- Processes that are requesting to enter their critical section are busy waiting (consuming processor time needlessly).
  - If critical sections are long, it would be more efficient to block processes that are waiting.

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Synchronization Solutions

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of locking:
  - Protecting critical regions via locks.
- Uniprocessors – could disable interrupts:
  - Currently running code would execute without preemption.
  - Generally too inefficient on multiprocessor systems:
    - Operating systems using this are not broadly scalable.
- Modern machines provide special atomic (non-interruptible) hardware instructions:
  - Either test memory word and set value at once.
  - Or swap contents of two memory words.

Interrupt Disabling

- On a Uniprocessor:
  - mutual exclusion is preserved but efficiency of execution is degraded: while in CS, we cannot interleave execution with other processes that are in RS.
- On a Multiprocessor:
  - mutual exclusion is not preserved:
    - CS is now atomic but not mutually exclusive (interrupts are not disabled on other processors).

```
Process Pi:
repeat
   disable interrupts
      critical section
   enable interrupts
      remainder section
forever
```

Special Machine Instructions

- Normally, access to a memory location excludes other access to that same location.
- Extension: designers have proposed machines instructions that perform 2 actions atomically (indivisible) on the same memory location (e.g., reading and writing).
- The execution of such an instruction is also mutually exclusive (even on Multiprocessors).

Solution to Critical Section Problem using Locks

- The general layout is of lock solution is:

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (TRUE);
```

Disadvantages of Special Machine Instructions

- Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time.
- No Progress (Starvation) is possible when a process leaves CS and more than one process is waiting.
- They can be used to provide mutual exclusion but need to be complemented by other mechanisms to satisfy the bounded waiting requirement of the CS problem.
- See next slide for an example.

Mutex Locks

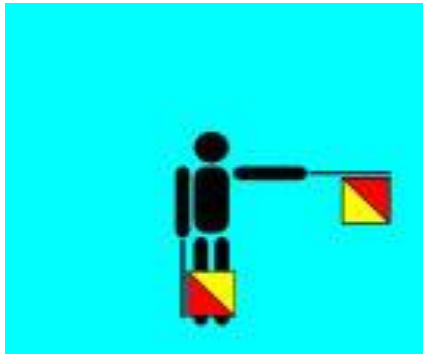
- Previous solutions are complicated and generally inaccessible to application programmers.
- OS designers build software tools to solve critical section problem.
- Simplest is mutex lock.
- Protect a critical section by first `acquire()` a lock then `release()` the lock:
  - Boolean variable indicating if lock is available or not.
- Calls to `acquire()` and `release()` must be atomic:
  - Usually implemented via hardware atomic instructions.
- But this solution requires busy waiting:
  - This lock therefore called a spinlock.

acquire() and release()

```
• acquire() {
        while (!available)
            ; /* busy wait */
        available = false;;
    }
•   release() {
        available = true;
    }
•   do {
    acquire lock
        critical section
    release lock
      remainder section
 } while (true);
```

Semaphores (1)

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Logically, a semaphore S is an integer variable that, apart from initialization, can only be changed through 2 atomic and mutually exclusive operations:
  - wait(S) (also down(S), P(S))
  - signal(S) (also up(S), V(S)
- Less complicated.



Critical Section of *n* Processes

- Shared data:

  semaphore mutex; // initiallized *to* 1

- Process *Pi:*

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (TRUE);
```

Semaphores (2)

- Access is via two atomic operations:

wait (*S*):

```
while (S <= 0);
S--;
```

signal (*S*):

```
S++;
```

Semaphores (3)

- Must guarantee that no 2 processes can execute wait() and signal() on the same semaphore at the same time.
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in CS 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.

Semaphores (4)

- To avoid busy waiting, when a process has to wait, it will be put in a blocked queue of processes waiting for the same event.
- Hence, in fact, a semaphore is a record (structure):

```
type semaphore = record
                   count: integer;
                   queue: list of process
                 end;
var S: semaphore;
```

- With each semaphore there is an associated waiting queue.
- Each entry in a waiting queue has 2 data items:
  - value (of type integer)
  - pointer to next record in the list

Semaphore's operations

- When a process must wait for a semaphore S, it is blocked and put on the semaphore's queue.
- Signal operation removes (assume a fair policy like FIFO) first process from the queue and puts it on list of ready processes.

```
wait(S):
  S.count--;
  if (S.count<0) {
    block this process
    place this process in S.queue
  }

signal(S):
  S.count++;
  if (S.count<=0) {
    remove a process P from S.queue
    place this process P on ready list
  }
```

Semaphore Implementation (1)

- Define semaphore as a C struct:

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

- Assume two operations:
  - **Block:** place the process invoking the operation on the appropriate waiting queue.
  - **Wakeup:** remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation (2)

- Semaphore operations now defined as

```
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);
        }
}
```

Two Types of Semaphores

1. *Counting* semaphore – integer value can range over an unrestricted domain.
2. *Binary* semaphore – integer value can range only between 0 and 1 (really a Boolean); can be simpler to implement (use waitB and signalB operations); same as Mutex lock.

- We can implement a counting semaphore *S* using 2 binary semaphores (that protect its counter).

Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0.
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of waiting processes.
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue (say, if LIFO) in which it is suspended.
- **Priority Inversion** – scheduling problem when lower-priority process holds a lock needed by higher-priority process

Problems with Semaphores

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes.
- But wait(S) and signal(S) are scattered among several processes. Hence, difficult to understand their effects.
- Usage must be correct in all the processes (correct order, correct variables, no omissions).
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- One bad (or malicious) process can fail the entire collection of processes.

SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY