CSCI-GA.2250-001

Operating Systems

Lecture 3:
Processes and Threads - Part 2

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

The basic idea is that the several components in any complex system will perform particular subfunctions that contribute to the overall function.

— *THE SCIENCES OF THE ARTIFICIAL,*

Herbert Simon

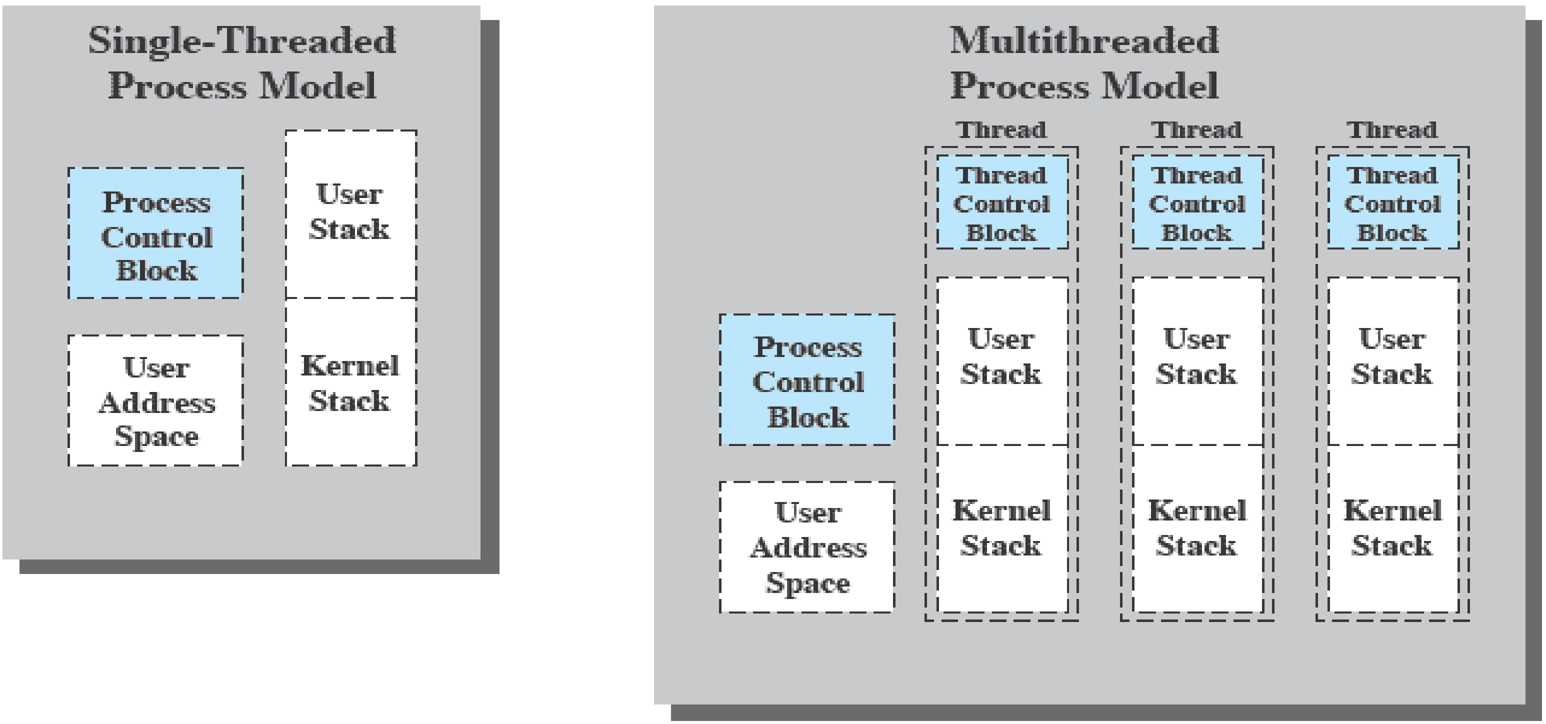
Processes Vs Threads

- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***
- The unit of resource ownership is referred to as a ***process*** or ***task***
- ***Multithreading*** - The ability of an OS to support multiple, concurrent paths of execution within a single process

Processes Vs Threads

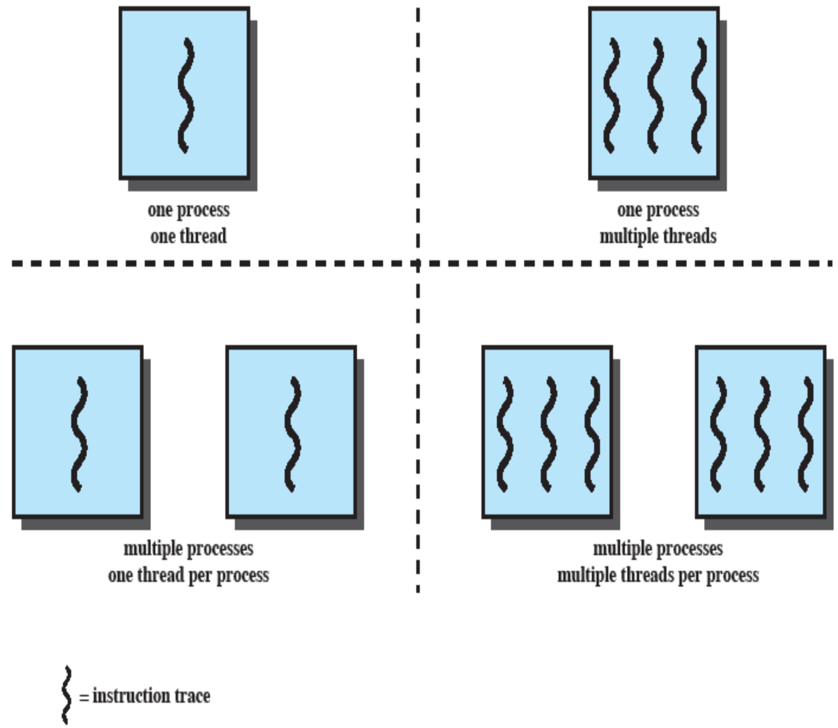
- Process is the unit for resource allocation and a unit of protection.
- Process has its own address space.
- A thread has:
  - an execution state (Running, Ready, etc.)
  - saved thread context when not running
  - an execution stack
  - some per-thread static storage for local variables
  - access to the memory and resources of its process (all threads of a process share this)

Processes Vs Threads



A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach … Example: MS-DOS

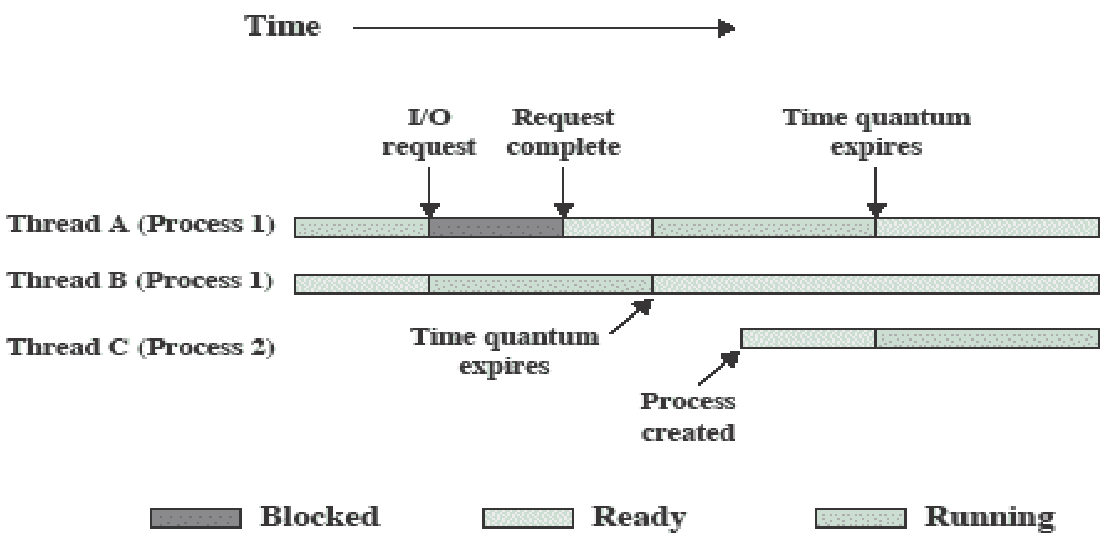
A Java run-time environment is an example of a system of one process with multiple threads.



Benefits of Threads

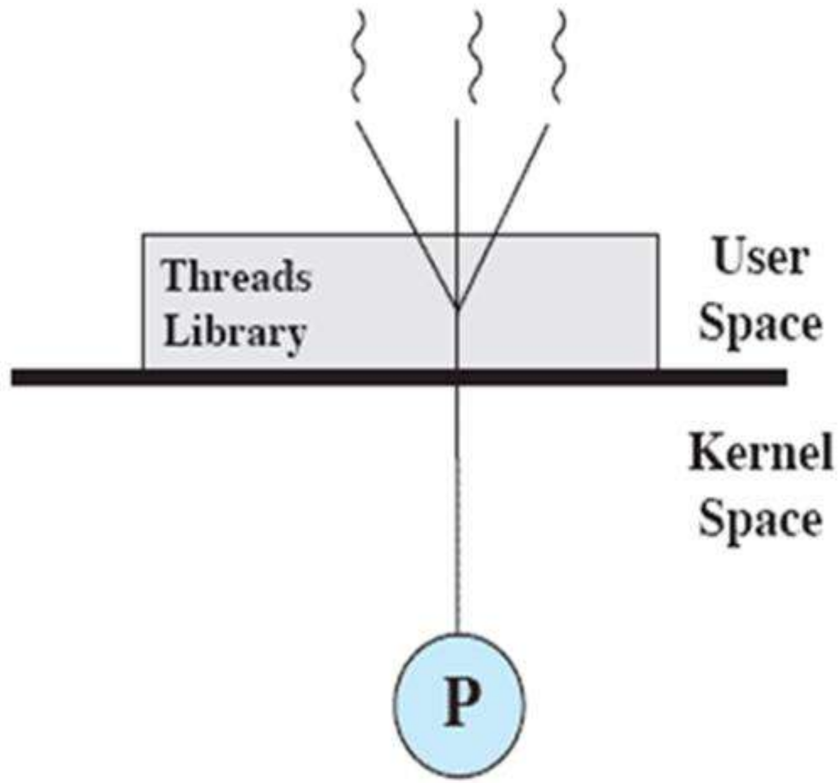
- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs

Multithreading on Uniprocessor System



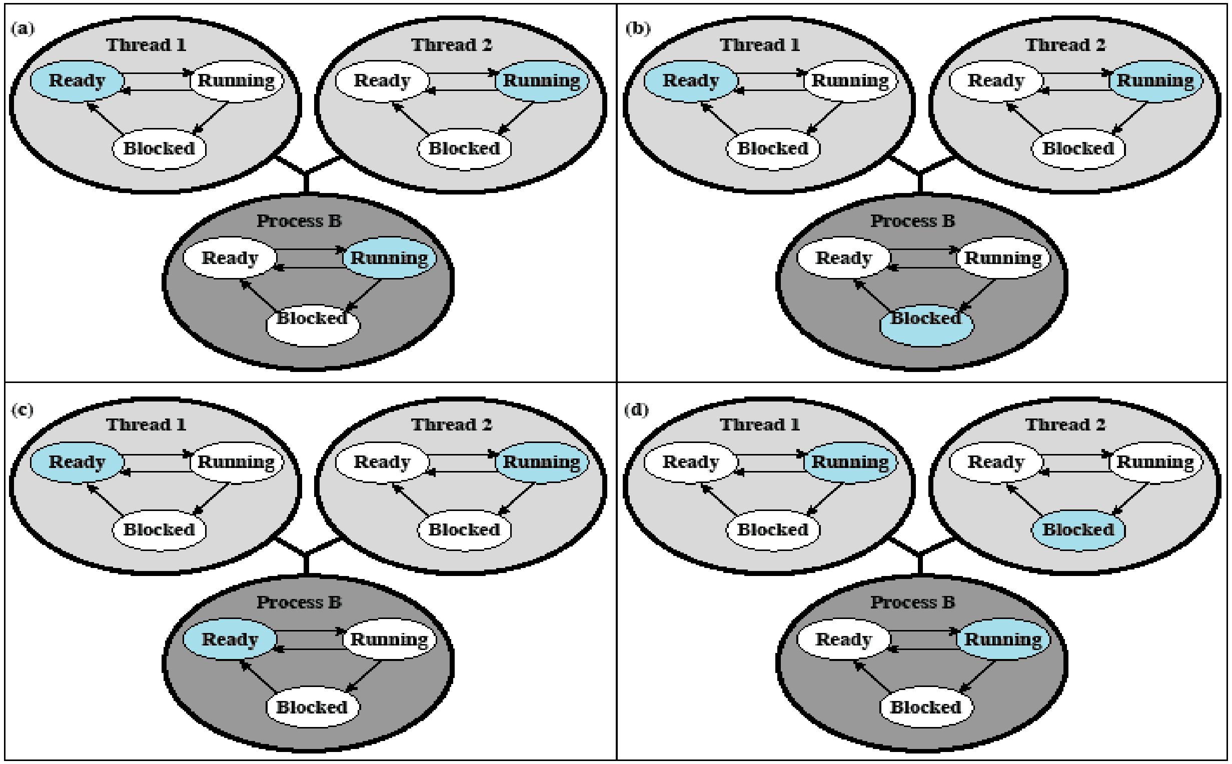
User-Lever Threads (ULT)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



User-Level Threads (ULTs)

- The kernel continues to schedule the process as a unit and assigns a single execution state .



Colored state
is current state

User-Level Threads (ULTs)

Advantages

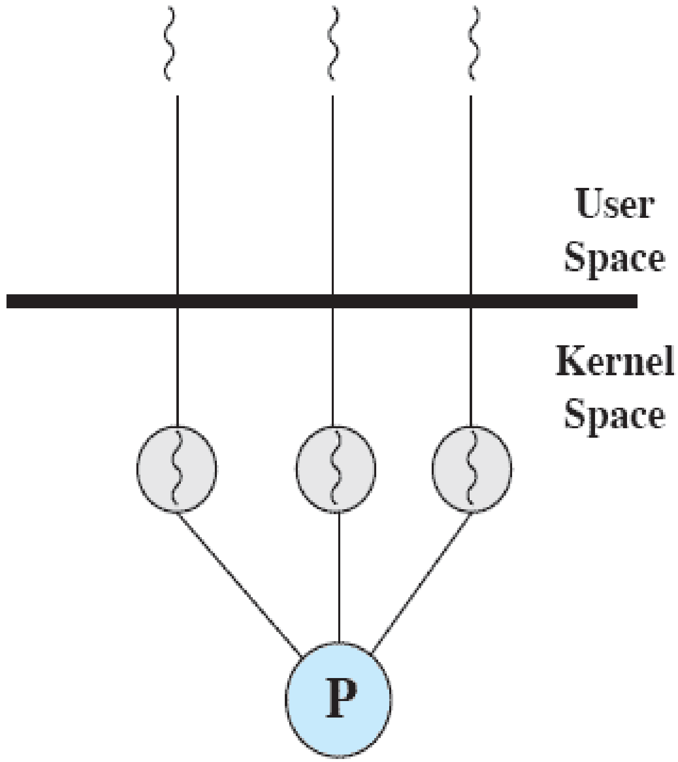
- Thread switch does not require kernel-mode.
- Scheduling (of threads) can be application specific.
- Can run on any OS.

Disadvantages

- A system-call by one thread can block all threads of that process.
- In pure ULT, multithreading cannot take advantage of multiprocessing

Kernel-Level Threads (KLTs)

- Thread management is done by the kernel
- no thread management is done by the application
- Windows OS is an example of this approach



Kernel-Level Threads (KLTs)

Advantages

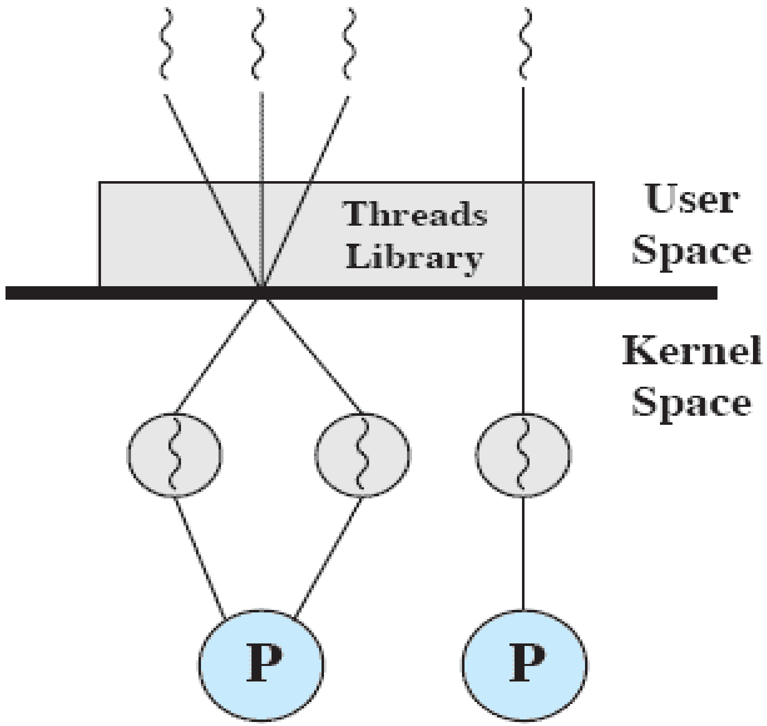
- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded

Disadvantages

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Combined (Hybrid) Approach

- Thread creation is done completely in user space.
- Bulk of scheduling and synchronization of threads is by the application (i.e. user space).
- Multiple ULTs from a single application are mapped onto (smaller or equal) number of KLTs.
- Solaris is an example



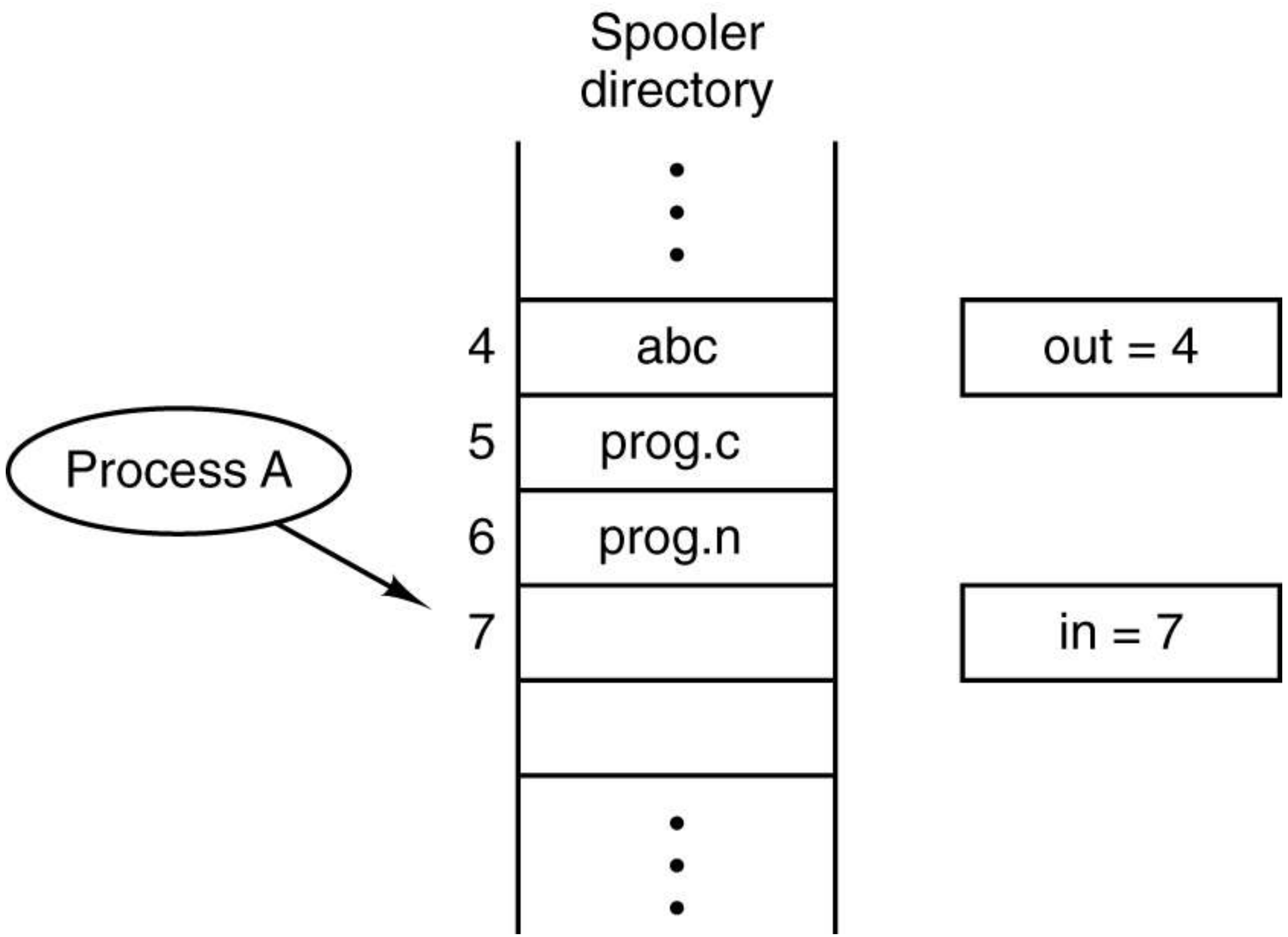
Threads and Processes Relationship

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

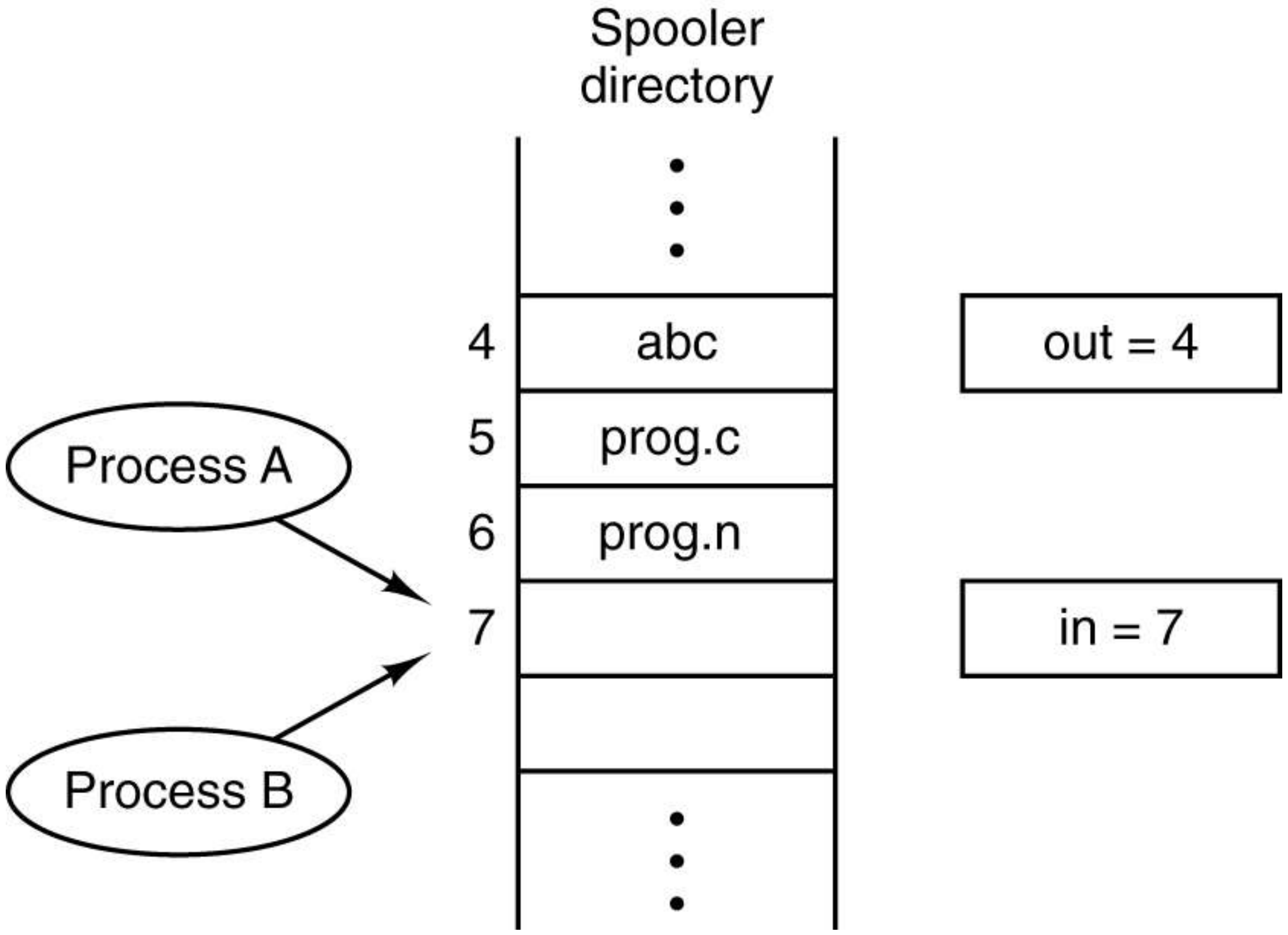
Interprocess Communication (IPC)

- Processes frequently need to communicate with other processes
- Three main issues:
  - How can one process pass information to another?
  - Need to make sure two or more processes do not get in each other's way.
  - Ensure proper sequencing when dependencies exist

Example of IPC

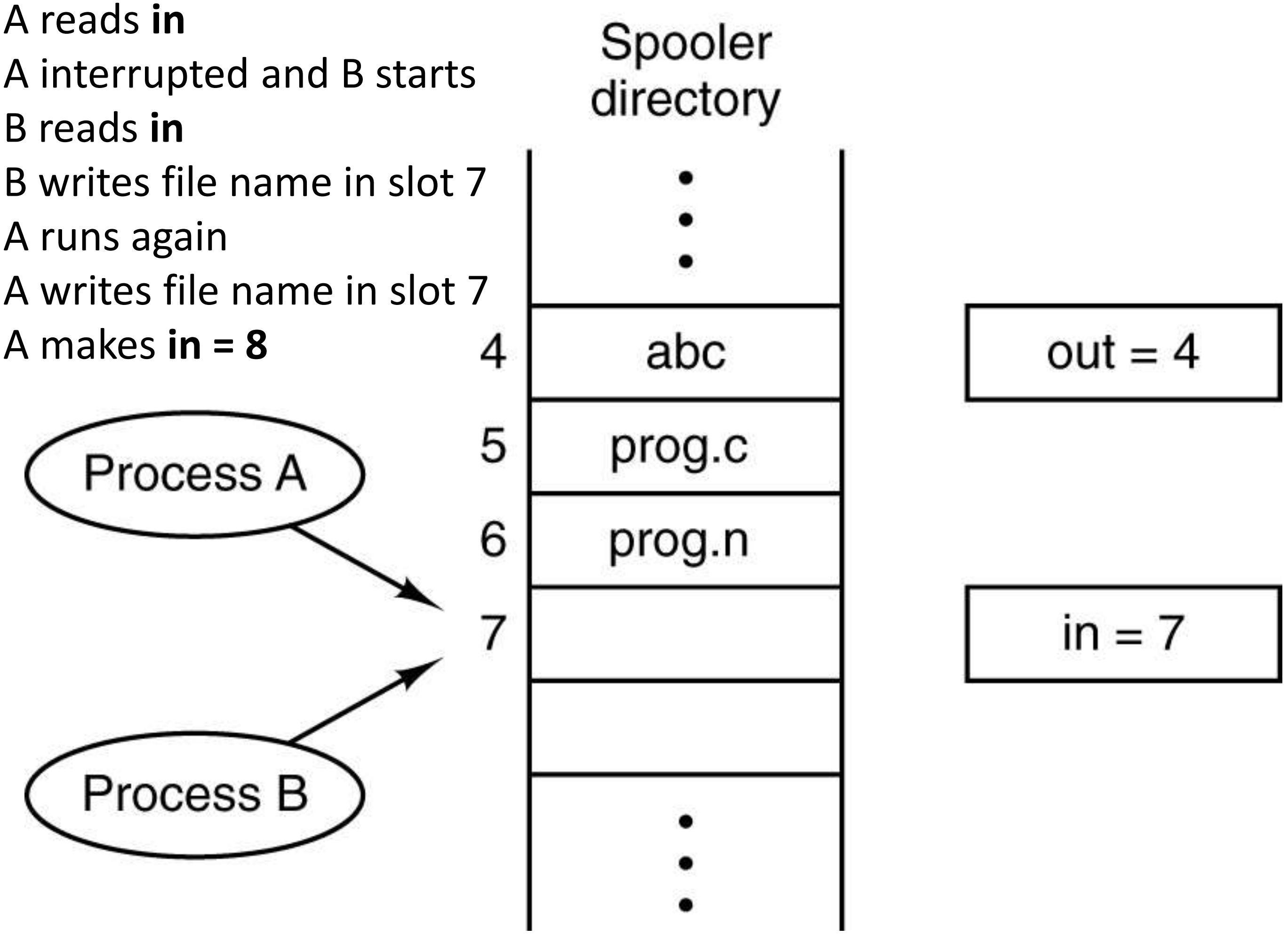


Example of IPC



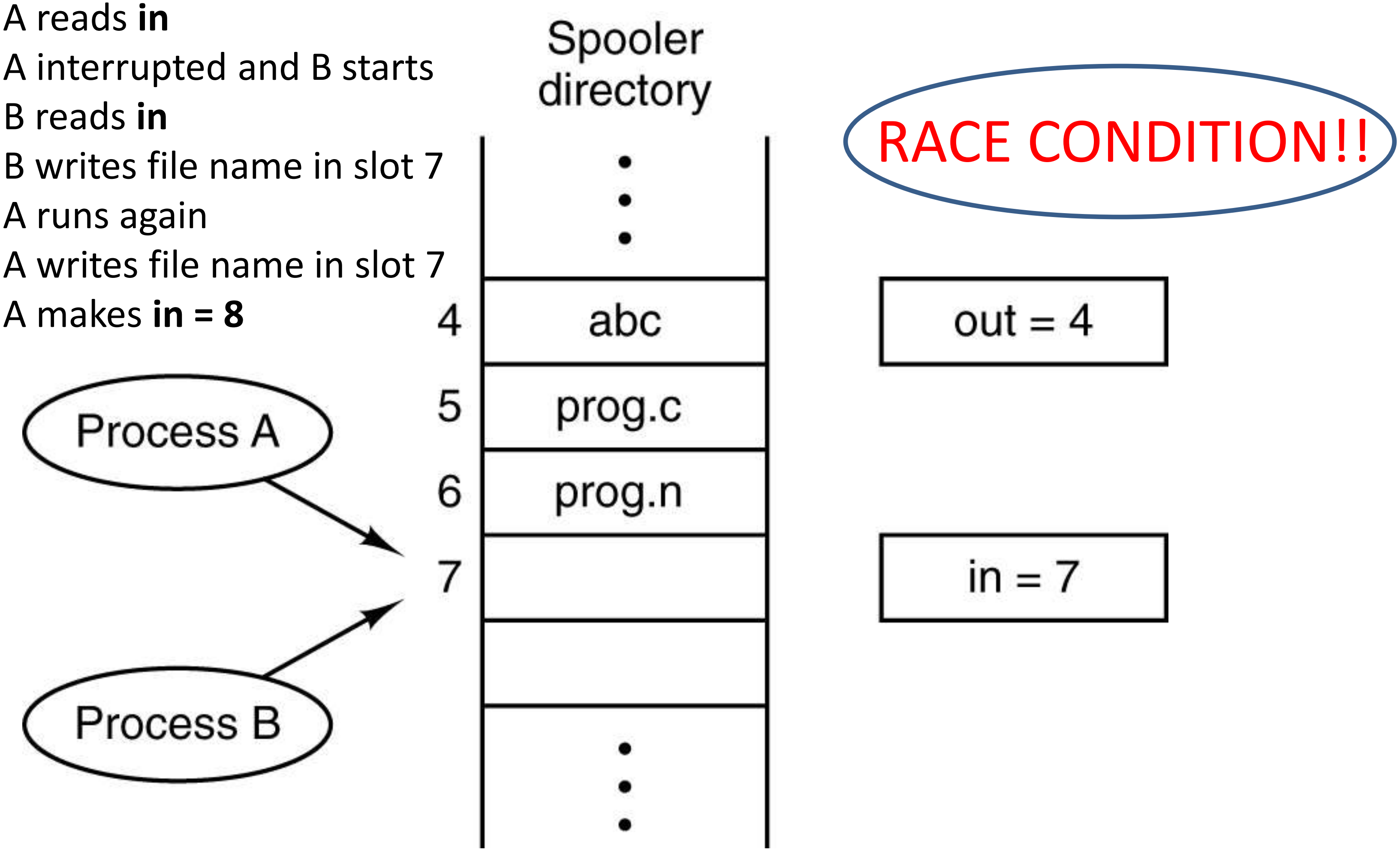
Example of IPC

1. Process A reads **in**
2. Process A interrupted and B starts
3. Process B reads **in**
4. Process B writes file name in slot 7
5. Process A runs again
6. Process A writes file name in slot 7
7. Process A makes **in = 8**



Example of IPC

1. Process A reads **in**
2. Process A interrupted and B starts
3. Process B reads **in**
4. Process B writes file name in slot 7
5. Process A runs again
6. Process A writes file name in slot 7
7. Process A makes **in = 8**

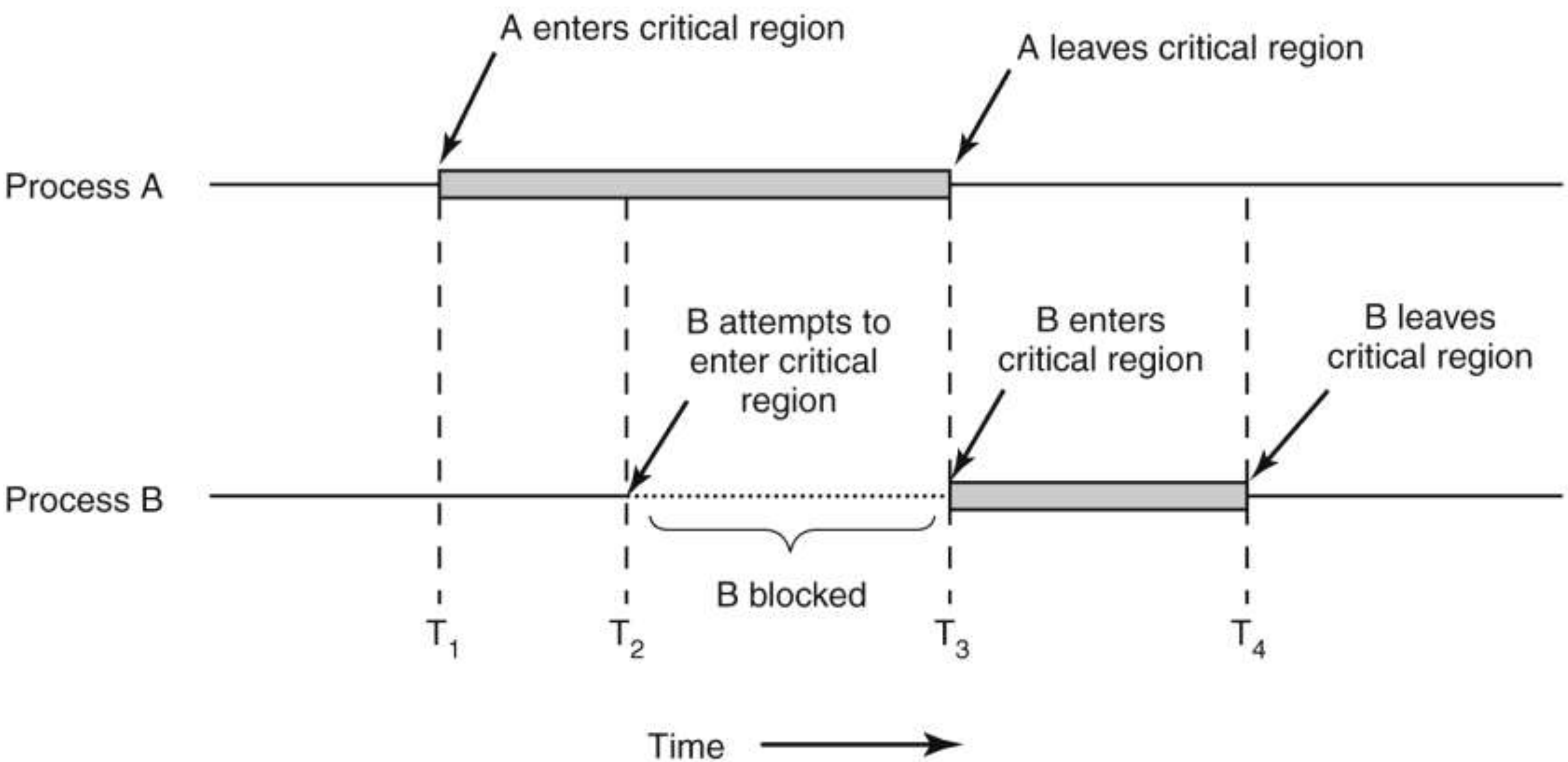


How to Avoid Race Condition?

- Prohibit more than one process from reading and writing the shared data at the same time -> mutual exclusion
- The choice of appropriate primitive operations for achieving mutual exclusion is a major design issue in an OS
- The part of the program where the shared memory is accessed is called the critical region

Conditions of Good Solutions

1. No two processes may be simultaneously inside their critical region
2. No assumptions may be made about speeds or the number of CPUs
3. No process running outside its critical region may block other processes
4. No process has to wait forever to enter its critical region


A enters critical region
A leaves critical region
Process A
B attempts to enter critical region
B enters critical region
B leaves critical region
Process B
B blocked
T1
T2
T3
T4
Time

Solution 1: Disabling Interrupts

Have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.

Solution 1: Why is it Bad?

- Unwise to give user processes the power to turn off interrupts
- Affects only one CPU and not other CPUs in the system in case of multicore or multiprocessor systems

Solution 2: Lock Variables

Have a shared (lock) variable, initially set to 0. When a process wants to enter its critical region, it first tests the lock:

- If 0, the process sets it to 1 and enters the critical region
- If 1, process waits until it becomes 0

Solution 2: Why is it Bad?

- Process A reads the lock and finds it 0
- Before it can set it to 1, process A is stopped and process B starts
- Process B finds the lock to be 0, so it sets it to 1 and enters the critical region
- Process B is stopped and process A runs
- Process A sets the lock to 1 and enters the critical region

Solution 2: Why is it Bad?

- Process A reads the lock and finds it 0
- Before it can set it to 1, process A is stopped and process B starts
- Process B finds the lock to be 0, so it sets it to 1 and enters the critical region
- Process B is stopped and process A runs
- Process A sets the lock to 1 and enters the critical region

Two processes will be in the critical region at the same time!!

Solution 3:
Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Process 0

Variable turn is initially 0

Process 1

Solution 3:
Strict Alternation

Busy waiting

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Variable turn is initially 0

Process 0

Process 1

Solution 3:
Strict Alternation: Why Bad?

What if process 0 is much faster than process 1?

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Process 1 spends a lot of time here!

Process 0

Process 1

Solution 3: Strict Alternation: Why Bad?

What if process 0 is much faster than process 1?

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Process 1 spends a lot of time here!

Process 0

Process 1

Violating condition 3!!

Taking turn is not a good idea when one of the processes is much slower than the other.

Solution 4: Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Hardware Solution

- The instruction: TSL RX, LOCK
  - TSL = Test and Set Lock
  - Reads the content of memory word *lock* into register RX, and then stores a nonzero value into *lock*
  - The whole operation is atomic

Hardware Solution

```
enter_region:
        TSL REGISTER,LOCK      | copy lock to register and set lock to 1
        CMP REGISTER,#0        | was lock zero?
        JNE enter_region       | if it was nonzero, lock was set, so loop
        RET                    | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0           | store a 0 in lock
        RET                    | return to caller
```

Similar Hardware Solution

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

About Previous Solutions

- Processes must call enter_region and leave_region in the correct timing. If a process cheats, the mutual exclusion will fail.
- The main drawbacks of all these solutions is busy waiting. Keeping the CPU busy doing nothing is not the best thing to do.
  - Wastes CPU time
  - Priority inversion problem

Sleep and Wakeup

- IPC primitives
- Block instead of wasting CPU time
- Two systemcalls:
  - sleep: causes the caller to block until another process wakes it up
  - wakeup: has one parameter, the process to be awakened

First Let's see the: Producer Consumer Problem

- Two processes share a common fixed size buffer
- One process (producer): puts info into the buffer
- The other process (consumer): removes info from the buffer

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count − 1;                    /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

```c
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item( );         /* generate next item */
                if (count == N) sleep( );       /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer); /* was buffer empty? */
        }
}
```

What happens if consumer() stopped after reading count (=0) ?

LOST WAKEUP PROBLEM

```c
void consumer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
                item = remove_item( );          /* take item out of buffer */
                count = count − 1;              /* decrement count of items in buffer */
                if (count == N − 1) wakeup(producer); /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

How to Solve The Lost Wakeup Problem?

- Add a wakeup waiting bit to the picture
  - When a wakeup is sent to a process that is still awake, this bit is set.
  - Later, when the process tries to go to sleep and the bit is set, the bit will be reset but the process will remain awake.
- BUT: What happens when we have more than two processes? How many bits shall we use?

Better Solution for Lost Wakeup Problem: **Semaphores**

- Integer to count the number of wakeups saved for future use
- Two primitives: down and up
  - atomic actions

down: if value = 0 then sleeps

otherwise, decrements it and continue

up: increments the value, and wakes up a sleeping process (if any)

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                  /* TRUE is the constant 1 */
                item = produce_item( );   /* generate something to put in buffer */
                down(&empty);           /* decrement empty count */
                down(&mutex);           /* enter critical region */
                insert_item(item);      /* put new item in buffer */
                up(&mutex);             /* leave critical region */
                up(&full);              /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                  /* infinite loop */
                down(&full);            /* decrement full count */
                down(&mutex);           /* enter critical region */
                item = remove_item( );    /* take item from buffer */
                up(&mutex);             /* leave critical region */
                up(&empty);             /* increment count of empty slots */
                consume_item(item);     /* do something with the item */
        }
}
```

Mutexes??

- A variable that can be in one of two states: locked and unlocked
- Can be used to manage critical sections
- Managed used TSL or XCHG

```
mutex_lock:
        TSL REGISTER,MUTEX        | copy mutex to register and set mutex to 1
        CMP REGISTER,#0           | was mutex zero?
        JZE ok                    | if it was zero, mutex was unlocked, so return
        CALL thread_yield         | mutex is busy; schedule another thread
        JMP mutex_lock            | try again
ok:     RET                       | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0             | store a 0 in mutex
        RET                       | return to caller
```

Didn't We Say Processes Do Not Share Address Space?

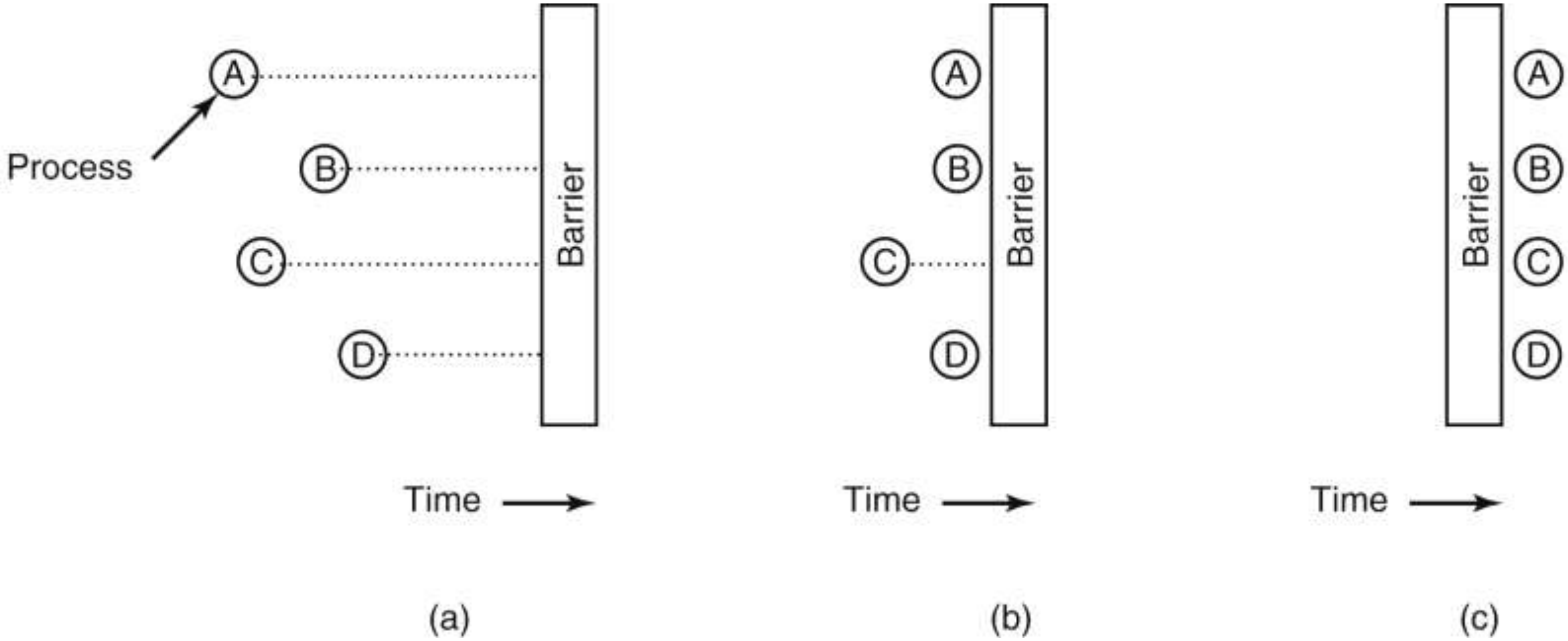
- Some of the shared data structures can be stored in the kernel and accessed through system calls
- Most modern OSes offer ways to processes to share some portions of their address spaces with other processes

Forget About Sharing: How About Message Passing?

- Two primitives: send and receive
- May be used across machines
- Are system calls
  - send(destination, &message)
  - receive(source, &message)
- Issues
  - Lost acknowledgement
  - Authentication
  - performance (message passing is always slower than stuff like semaphores, …)

Barriers

- Synchronization mechanisms
- Intended for group of processes

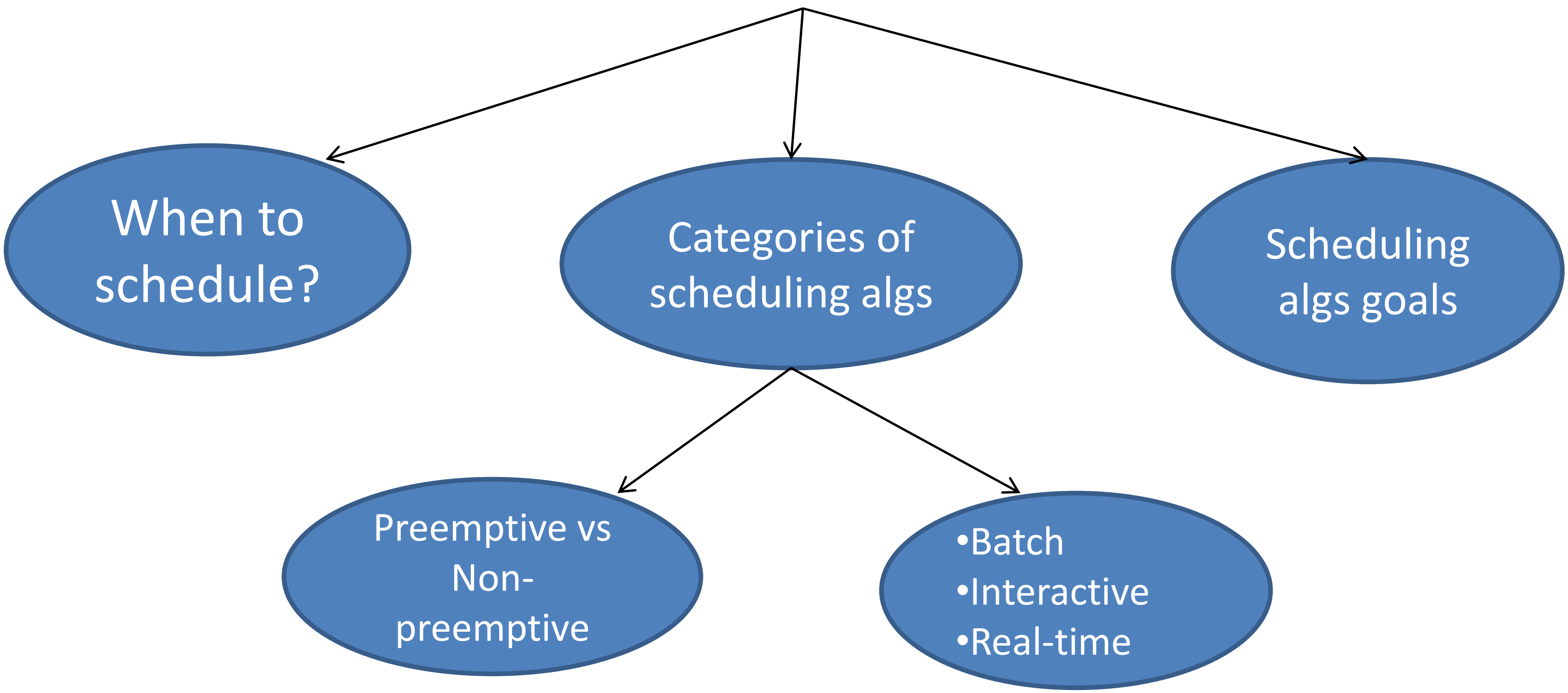


Scheduling

Given a group of ready processes, which process to run?

Scheduling

Given a group of ready processes, which process to run?



When to Schedule?

- When a process is created
- When a process exits
- When a process blocks
- When an I/O interrupt occurs

Categories of Scheduling Algorithms

- Batch
  - No users impatiently waiting
  - mostly nonpreemptive, or preemptive with long period for each process
- Interactive
  - preemption is essential
- Real-time
  - deadlines

Scheduling Algorithms Goals

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Scheduling in Batch Systems: First-Come First-Served

- Nonpreemptive
- Processes ordered as queue
- A new process added to the end of the queue
- A blocked process that becomes ready added to the end of the queue
- Main disadv: Can hurt I/O bound processes

Scheduling in Batch Systems: Shortest Job First

- Nonpreemtive
- Assumes runtime is known in advance
- Is only optimal when all the jobs are available simultaneously

Run in original order

Run in shortest job first

Scheduling in Batch Systems: Shortest Remaining Time Next

- Preemptive
- Scheduler always chooses the process whose remaining time is the shortest
- Runtime has to be known in advance

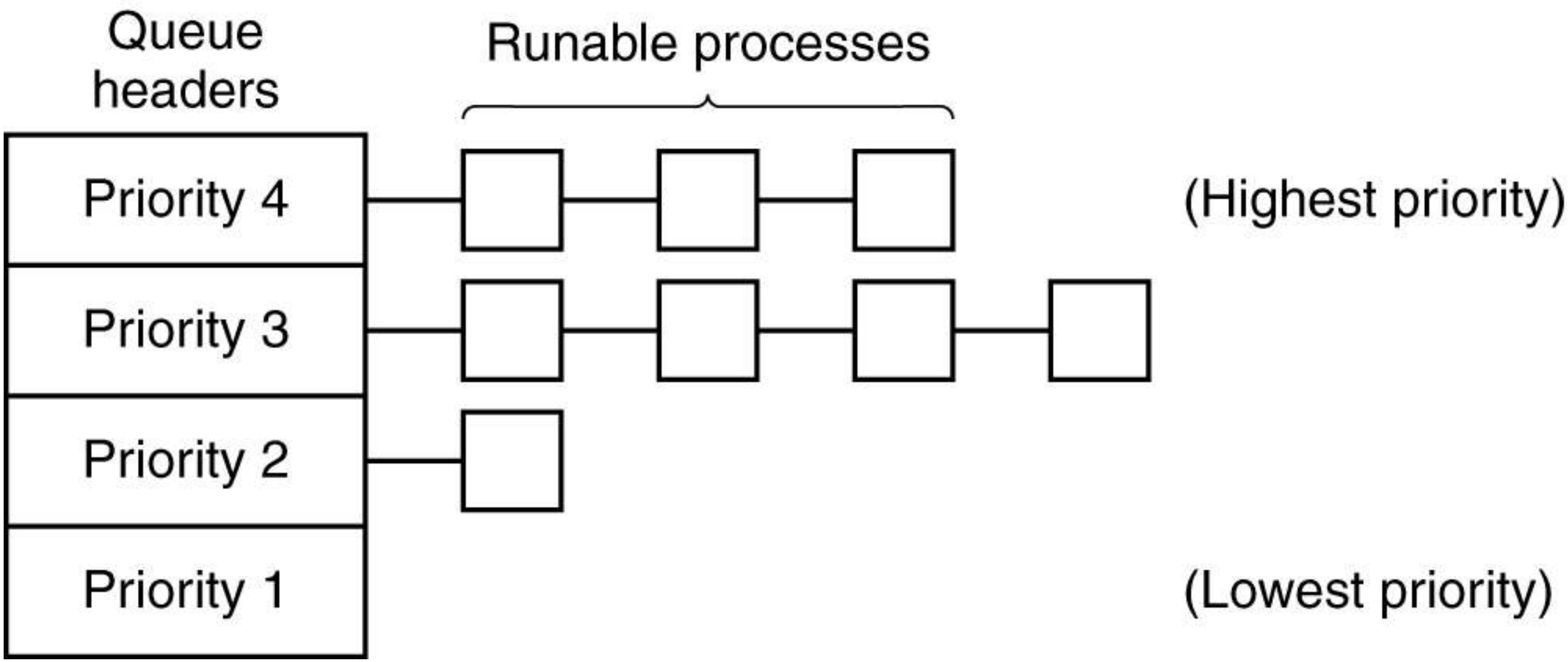
Scheduling in Interactive Systems: Round-Robin

- Each process is assigned a time interval: quantum
- After this quantum, the CPU is given to another process
- What is the length of this quantum?
  - too short -> too many context switches -> lower CPU efficiency
  - too long -> poor response to short interactive
  - quantum longer than CPU burst is good (why?)

Scheduling in Interactive Systems: Priority Scheduling

- Each process is assigned a priority
- runnable process with the highest priority is allowed to run
- Priorities are assigned statically or dynamically
- Must not allow a process to run forever
  - Can decrease the priority of the currently running process
  - Use time quantum for each process

Scheduling in Interactive Systems: Multiple Queues



Scheduling in Interactive Systems: Other Schemes

- Shortest process next
  - Estimate running time based on past behavior
- Guaranteed schedule
  - Make promise to the user and live up to the promise
- Lottery scheduling
  - Give each process one or more lottery tickets
- Fair-Share scheduling
  - Take the user into account

Scheduling in Real-Time

- Process must respond to an event within a deadline
- Hard real-time vs soft real-time
- Periodic vs aperiodic events
- Processes must be schedulable
- Scheduling algorithms can be static or dynamic

Thread Scheduling

- Two levels of parallelism: processes and threads within processes
- Kernel-bases vs user-space

Conclusion

- Threads and processes are crucial concepts in OS design.
- As OS designer, you must make decision regarding: process table, threading, scheduling, etc.
- We have covered more stuff than the book so you may find information here more than the book (especially in mutual exclusion part).