Inter-process Communication

Operating Systems

Review of last lecture

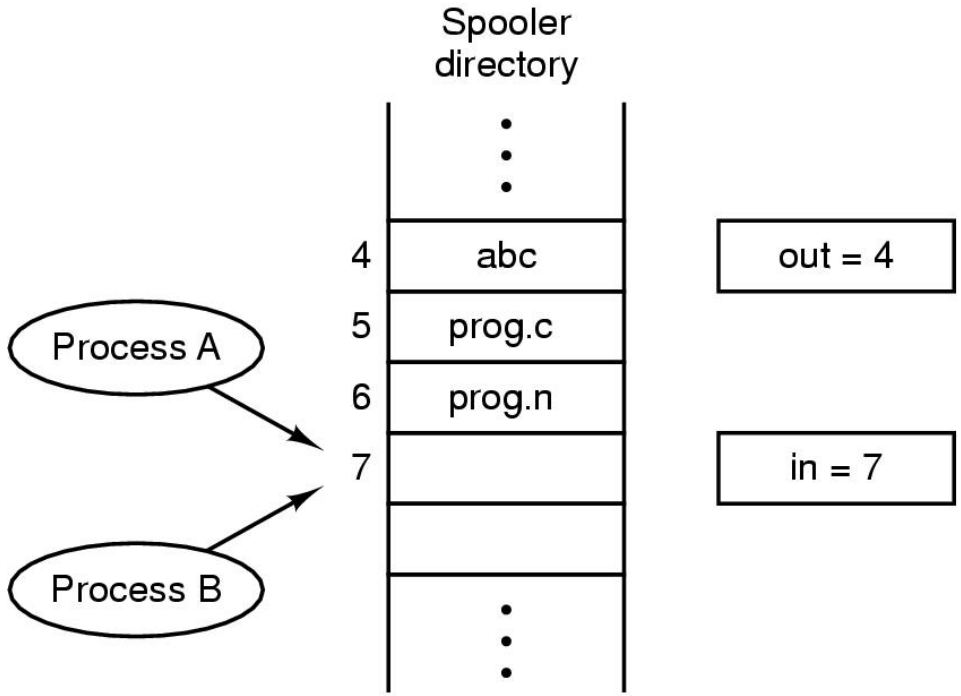
- Multithreading is very useful for applications
  - Efficient multithreading requires fast primitives
  - Processes are too heavyweight
- Solution is to separate threads from processes
  - Kernel-level threads much better, but still significant overhead
  - User-level threads even better, but not well integrated with OS
- Now, how do we get our threads to correctly cooperate with each other?

Inter-process Communication (IPC)

- Processes frequently need to communicate with other processes
  - To pass information from process to another
  - To make sure two or more processes do not get in each other's way
  - To enable proper sequencing when dependencies are present

Race Conditions

- Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
- Ex. two processes want to access the printer spooler directory at the same time



Conditions required to avoid race condition

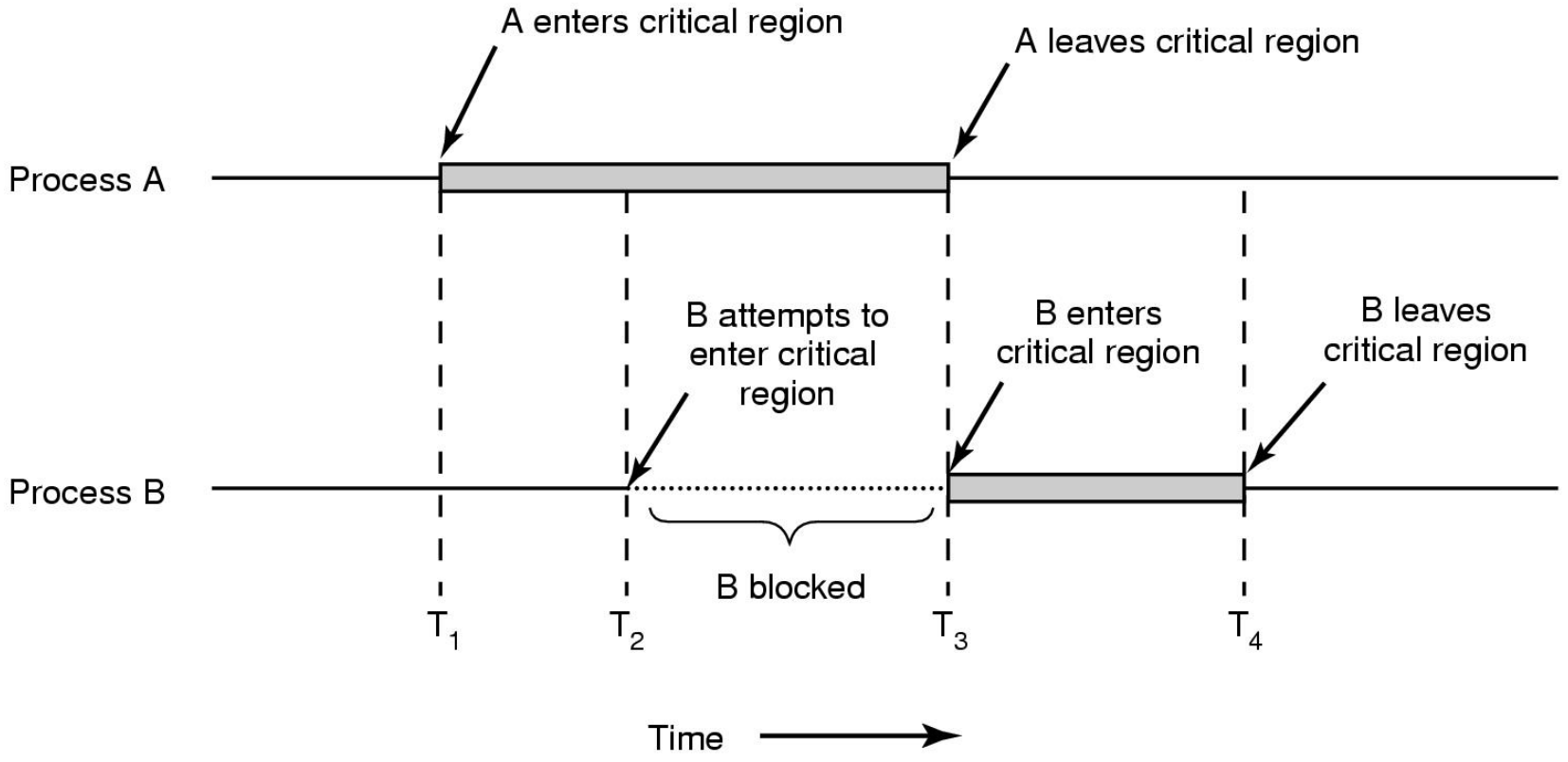
1. No two processes may be simultaneously inside their critical regions. – Mutual Exclusion (mutex)
2. No assumptions may be made about speeds or the number of CPUs. – No Assumption
3. No process running outside its critical region may block other processes. – Progress
4. No process should have to wait forever to enter its critical region. – No Starvation

Mutual Exclusion

- We want to use mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks
- Code that uses mutual exclusion to synchronize its execution is called a critical region (or critical section)
  - Only one thread at a time can execute in the critical region
  - All other threads are forced to wait on entry
  - When a thread leaves a critical region, another can enter
  - Example: sharing your bathroom with housemates

Mutual Exclusion using Critical Regions

- Critical region – the part of the program where shared variables are accessed



Critical Region Requirements *(apply to both thread and process)*

- Mutual exclusion (mutex)
  - No other thread must execute within the critical region while a thread is in it
- Progress
  - A thread in the critical region will eventually leave the critical region
  - If some thread T is not in the critical region, then T cannot prevent some other thread S from entering the critical region
- Bounded waiting (no starvation)
  - If some thread T is waiting on the critical region, then T should only have wait for a bounded number of other threads to enter and leave the critical region
- No assumption
  - No assumption may be made about the speed or number of CPUs

Mechanisms for Building Critical Regions

- To ensure that no two processes are ever in their critical regions at the same time.
  - Atomic read/write
    - Alternation
  - Locks
    - Primitive, minimal semantics, used to build others
  - Semaphores
    - Basic, easy to get the hang of, but hard to program with
  - Monitors
    - High-level, requires language support, operations implicit
  - Message Passing
    - System calls to send and receive messages between processes

Mutual Exclusion with Busy Waiting

- Disabling interrupts
  - Each process disables all interrupts just after entering its critical region and re-enable them just before leaving it
  - In a multicore (i.e., multiprocessor system) disabling the interrupts of one CPU does not prevent other CPUs from interfering with operations the first CPU is performing
- Lock variables
  - A single, shared (lock) variable, process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.
  - Has the same fatal flaw that we saw in the spooler directory.

Mutual Exclusion with Busy Waiting

- Strict alternation
  - two processes strictly alternate in entering their critical regions
  - it is not really a serious candidate as a solution because it violates condition 3
- Peterson's solution
- The TSL instruction

Semaphores

- Semaphores are an *abstract data* type that provide mutual exclusion to critical region
- Semaphores are *integers* that support two operations:
  - wait(semaphore): decrement, block until semaphore is open – Down
  - signal(semaphore): increment, allow another thread to enter – Up
- Down : checks the value of the semaphore
  - If value is > 0, then it decrements it (by using one wakeup signal) and continues
  - If value = 0, then the process is put to sleep without completing its down operation
- Up : increments the value of the semaphore
  - If there are processes sleeping on the semaphore, then one of them is chosen, and it is allowed to complete its down operation
- Checking the value of a semaphore and updating them is done in an atomic fashion

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes/threads
- When Down() is called by a thread:
  - If semaphore is open (> 0), thread continues
  - If semaphore is closed, thread blocks on queue
- Then Up() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
    - What if multiple threads are waiting on the queue?
  - If no threads are waiting on the queue, the signal is remembered for the next thread
    - In other words, V() has “history” (c.f., condition vars later)
    - This “history” is a counter

Monitors

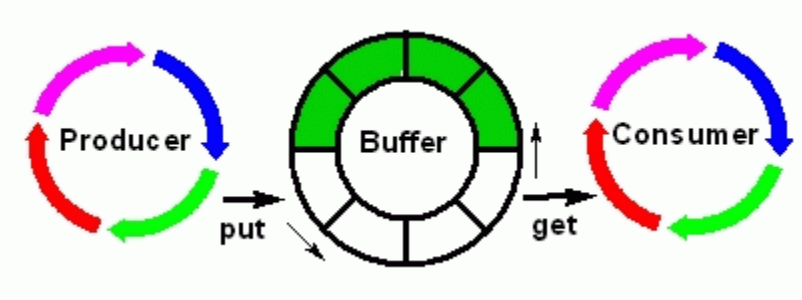
- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
- A monitor protects its data from unstructured access
  - It guarantees that threads accessing its data through its procedures interact only in legitimate ways
- For ensuring mutual exclusion, programmer turns all the critical regions into monitor procedures, this way no two process can enter their critical region at the same time
- When a process calls a monitor procedure:
  - The first couple of instructions of the procedure will check to see if any other process is currently using the monitor
  - If so the calling process will suspended until the other process leaves the monitor
  - If no process is using the monitor, the calling process may enter

Message Passing

- Uses two system calls, send and receive
  - send(destination, &message);
  - receive(source, &message);
- Simple model of communication and synchronization based on atomic transfer of data across a channel
- Commonly used in parallel programming systems
- Direct application to distributed systems

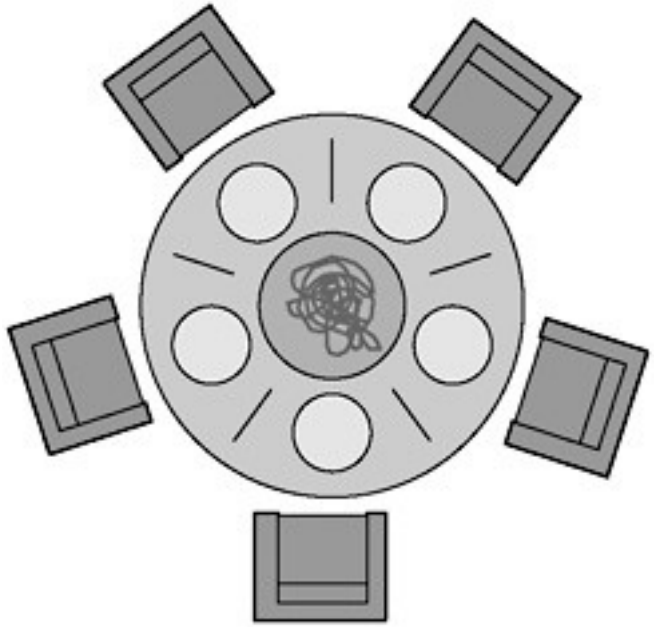
Classical IPC Problems

- Producer–consumer problem
  - Models access to a bounded-buffer
  - Producer won't try to add data into the buffer if it's full
  - Consumer won't try to remove data from an empty buffer



Classical IPC Problems

- Dining philosophers problem (Dijkstra)
  - Models processes competing for exclusive access to a limited number of resources such as I/O devices



*Classical IPC Problems

- Readers–writers problem
  - Models access to a database
  - Two readers can read at once
  - A writer should not wait longer than needed
  - Fairness for both readers and writers