Process Management

Topics

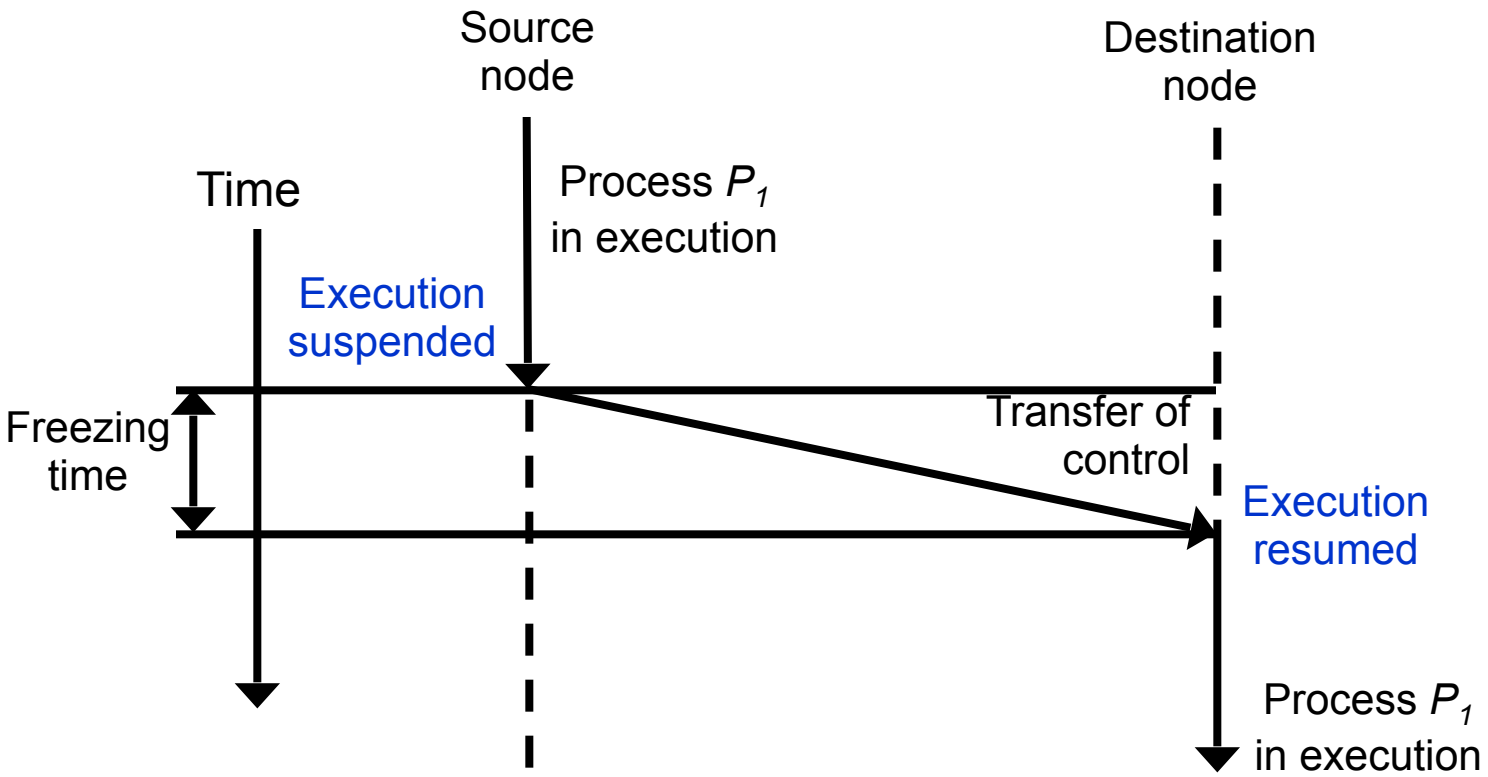
- Process Migration
  - Desired features of a good process migration mechanism
  - Process migration mechanisms
  - Process migration in heterogeneous systems
  - Advantage of process migration
- Threads
  - Motivation for using threads
  - Models for organizing threads
  - Issues in designing a thread package
  - Implementing a thread package

Introduction

- The main goal of process management is to make the best possible use of the processing resources of the entire system by sharing them among all processes.
- Three important concepts are used:
  - *Process allocation* deals with the process of deciding which process should be assigned to which processor.
  - *Process migration* deals with the movement of a process from its current location to the processor to which it has been assigned
  - *Threads* deals with fine-grained parallelism for better utilization of the processing capability of the system.
- Process allocation has been introduced in previous chapter.

Flow of Execution of a Migrating Process

- Process migration is the relocation of a process from its current location (the *source node*) to another node (the *destination node*).



Type of Process Migration

- Non-preemptive process migration
  - A process is migrated before it starts executing on its source node.
- Preemptive process migration
  - A process is migrated during the course of its execution.

Steps of Process Migration

1. Selection of a process that should be migrated
2. Selection of the destination node to which the selected process should be migrated
3. Actual transfer of the selected process to the destination node

- The first two steps are taken care of by the process migration policy and the third step is taken care of by the process migration mechanism.
  - The process migration policies for selection have been introduced in the previous chapter.

Desirable Features of a Good Migration Mechanism

- Transparency
  - Object access level
    - It is the minimum requirement for a system to support non-preemptive process migration facility.
    - Access to objects such as files and devices can be done in a location-independent manner.
    - It allows free initiation of programs at an arbitrary node.
    - The system must provide a mechanism for transparent object naming and locating.
  - System call and inter-process communication level
    - It must be provided to support preemptive process migration.
    - However, system calls to request the physical properties of a node need not be location independent.
    - Transparency of IPC is also desired, that is, once a message has been sent, it should reach its receiver process without the need for resending it from the sender node.

Desirable Features of a Good Migration Mechanism

- Minimal Interference
  - One method to achieve this is by minimizing the *freezing time* of the process being migrated.
- Minimal Residual Dependencies
  - No residual dependency should be left on the previous node.
  - Otherwise,
    - The migrated process continues to impose a load on its previous node.
    - A failure or reboot of the previous node will cause the process to fail.

Desirable Features of a Good Migration Mechanism

- Efficiency
  - The main sources of inefficiency involved with process migration are as follows:
    - The time required for migrating a process
    - The cost of locating an object (including the migrated process)
    - The cost of supporting remote execution once the process is migrated
- Robustness
  - The failure of a node other than the one on which a process is currently running should not in any way affect the accessibility or execution of that process.

Desirable Features of a Good Migration Mechanism

- Communication between co-processes of a job
  - One further exploitation of process migration is the *parallel processing* among the processes of a single job distributed over several nodes.
  - If the facility is supported, to reduce communication cost, it is also necessary that those coprocesses be able to directly communicate with each other irrespective of their locations.

Subactivities of Process Migration

Subactivities of Process Migration

Address space

Communication

Subactivities of Process Migration

- The four major subactivities involved in process migration are as follows:
  1. Freezing the process on its source node and restarting it on its destination node
  2. Transferring the process's address space from its source node to its destination node
  3. Forwarding messages meant for the migrant process
  4. Handling communication between cooperating processes that have been separated (placed on different nodes) as a result of process migration

Mechanisms for Freezing and Restarting a Process

- In preemptive process migration, the usual process is to take a "snapshot" of the process's state on its source node and reinstate the snapshot on the destination node.

- The process is frozen on its source node, its state information is transferred to its destination node, and the process is restarted on its destination node using this state information.

Mechanisms for Freezing and Restarting a Process

Q: Under which condition, the process can be blocked, i.e., stopped for process migration?

Mechanisms for Freezing and Restarting a Process

- Immediate and Delayed Blocking of the Process
  - Before a process can be frozen, its execution must be blocked.
  - Depending upon the process's current state, it may be blocked immediately or the blocking may have to be delayed until the process reaches a state when it can be blocked.
  - Typical cases:

Mechanisms for Freezing and Restarting a Process

- Immediate and Delayed Blocking of the Process
  - Before a process can be frozen, its execution must be blocked.
  - Depending upon the process's current state, it may be blocked immediately or the blocking may have to be delayed until the process reaches a state when it can be blocked.
  - Typical cases:
    - If the process is not executing a system call, it can be immediately blocked from further execution.

Mechanisms for Freezing and Restarting a Process

- Immediate and Delayed Blocking of the Process
  - Before a process can be frozen, its execution must be blocked.
  - Depending upon the process's current state, it may be blocked immediately or the blocking may have to be delayed until the process reaches a state when it can be blocked.
  - Typical cases:
    - If the process is not executing a system call, it can be immediately blocked from further execution.
    - If the process is executing a system call but is sleeping at an interruptible priority waiting for a kernel event to occur, it can be immediately blocked.

Mechanisms for Freezing and Restarting a Process

- Immediate and Delayed Blocking of the Process
  - Before a process can be frozen, its execution must be blocked.
  - Depending upon the process's current state, it may be blocked immediately or the blocking may have to be delayed until the process reaches a state when it can be blocked.
  - Typical cases:
    - If the process is not executing a system call, it can be immediately blocked from further execution.
    - If the process is executing a system call but is sleeping at an interruptible priority waiting for a kernel event to occur, it can be immediately blocked.
    - If the process is executing a system call and is sleeping at a noninterruptible priority waiting for a kernel event to occur, the process's blocking has to be delayed until the system call is complete.

Mechanisms for Freezing and Restarting a Process

- Fast and Slow I/O Operations
  - After the process has been blocked, the next step in freezing the process is to wait for the completion of all fast I/O operations associated with process.
  - However, it is not feasible to wait for slow I/O operations, such as those on a pipe or terminal.
  - Theses slow I/O operations should be continued after the process starts executing on its destination node.

Mechanisms for Freezing and Restarting a Process

- Processes may open (remote or local) files for read and write.
- How can the processes continue to read and write on the opened files after process migration?

Mechanisms for Freezing and Restarting a Process

- Processes may open (remote or local) files for read and write.
- How can the processes continue to read and write on the opened files after process migration?
  - File Migration

Mechanisms for Freezing and Restarting a Process

- Processes may open (remote or local) files for read and write.
- How can the processes continue to read and write on the opened files after process migration?
  - File Migration
  - File State Migration

Mechanisms for Freezing and Restarting a Process

- Information about Open Files
  - A process's state information also consists of the information pertaining to files currently opened by the process.
  - In a distributed system that provides a network transparent execution environment, there is no problem in collecting this state information because the same protocol is used to access local as well as remote files using the system-wide unique file identifiers.
  - However, other systems identify files by their full pathnames, but a pathname loses significance once a file has been opened by a process because the OS returns to the process a file descriptor.

Mechanisms for Freezing and Restarting a Process

- Information about Open Files (Cont.)
  - Approaches to the systems using file descriptors:
    1. A link is created to the file and the pathname of the link is used as an access point to the file after the process migrates.
    2. An open file's complete pathname is reconstructed when required. For this, necessary modification have to be incorporated in the UNIX kernel.
  - Another file system issue is that one or more files being used by the process on its source node may also be present on its destination node.
    - It would be more efficient to access these files from the local node at which the process is executing.

Mechanisms for Freezing and Restarting a Process

- Reinstating the Process on its Destination Node
  - On the destination node, an empty process state is created that is similar to that allocated during process creation.
    - The newly allocated process may or may not have the same process identifier as the migrating process.
  - Once all the state of the migrating process has been transferred from the source node to the destination node and copied into the empty process state, the new copy of the process is unfrozen and the old copy is deleted.
  - Several special cases may require special handling and hence more work.
    - E.g., when obtaining the snapshot, the process may have been executing a system call since some calls are not atomic. How?

Address Space Transfer Mechanisms

- A process consists of the program being executed, along with the program's data, stack, and state.
- Process's state (KB)
  - which consists of the execution status (registers), scheduling information, memory tables, I/O states, capability list, process's identifier, process's user and group identifiers, information about the files opened by the process, and so on.
- Process's address space (MB)
  - code, data, and stack of the program.

Address Space Transfer Mechanisms

- The process's address space can be transferred to the destination node either before or after the process starts executing on the destination node.
- What may affect your decision for when to transfer the address space?

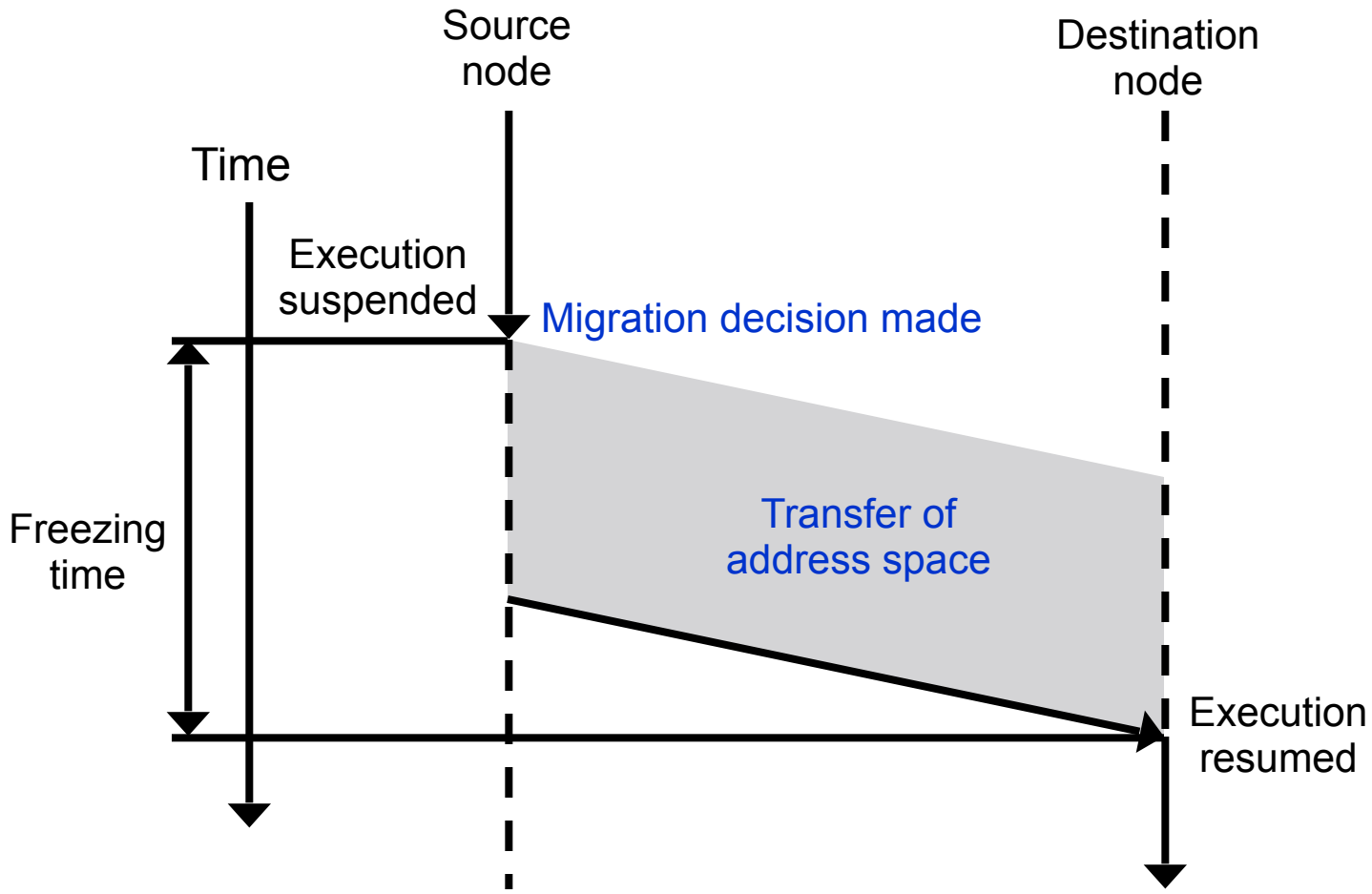
Address Space Transfer Mechanisms

- The process's address space can be transferred to the destination node either before or after the process starts executing on the destination node.

- What may affect your decision for when to transfer the address space?
  - Interactiveness
  - Size of the address space
  - Network bandwidth

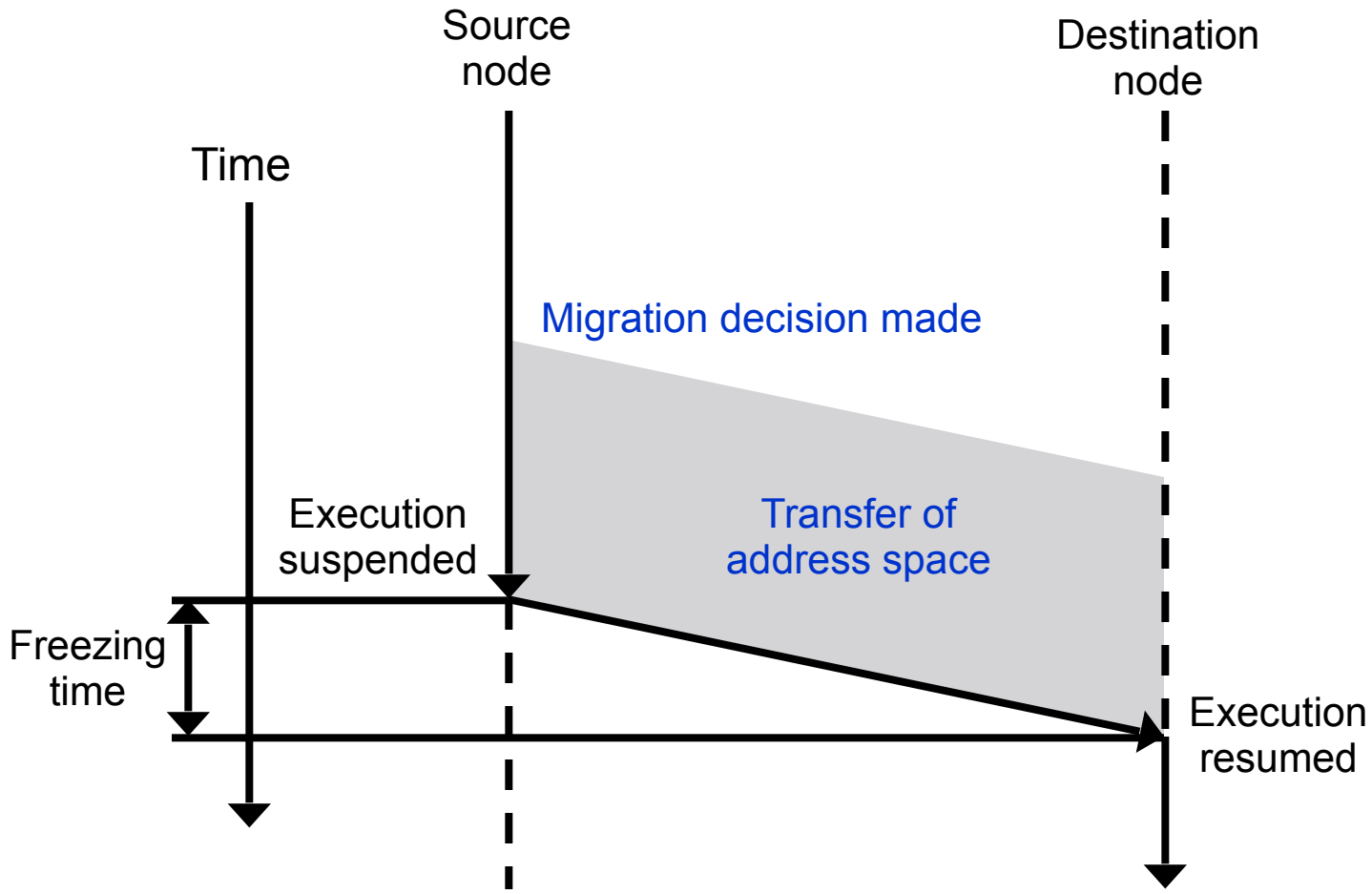
Address Space Transfer Mechanisms

- Total Freezing



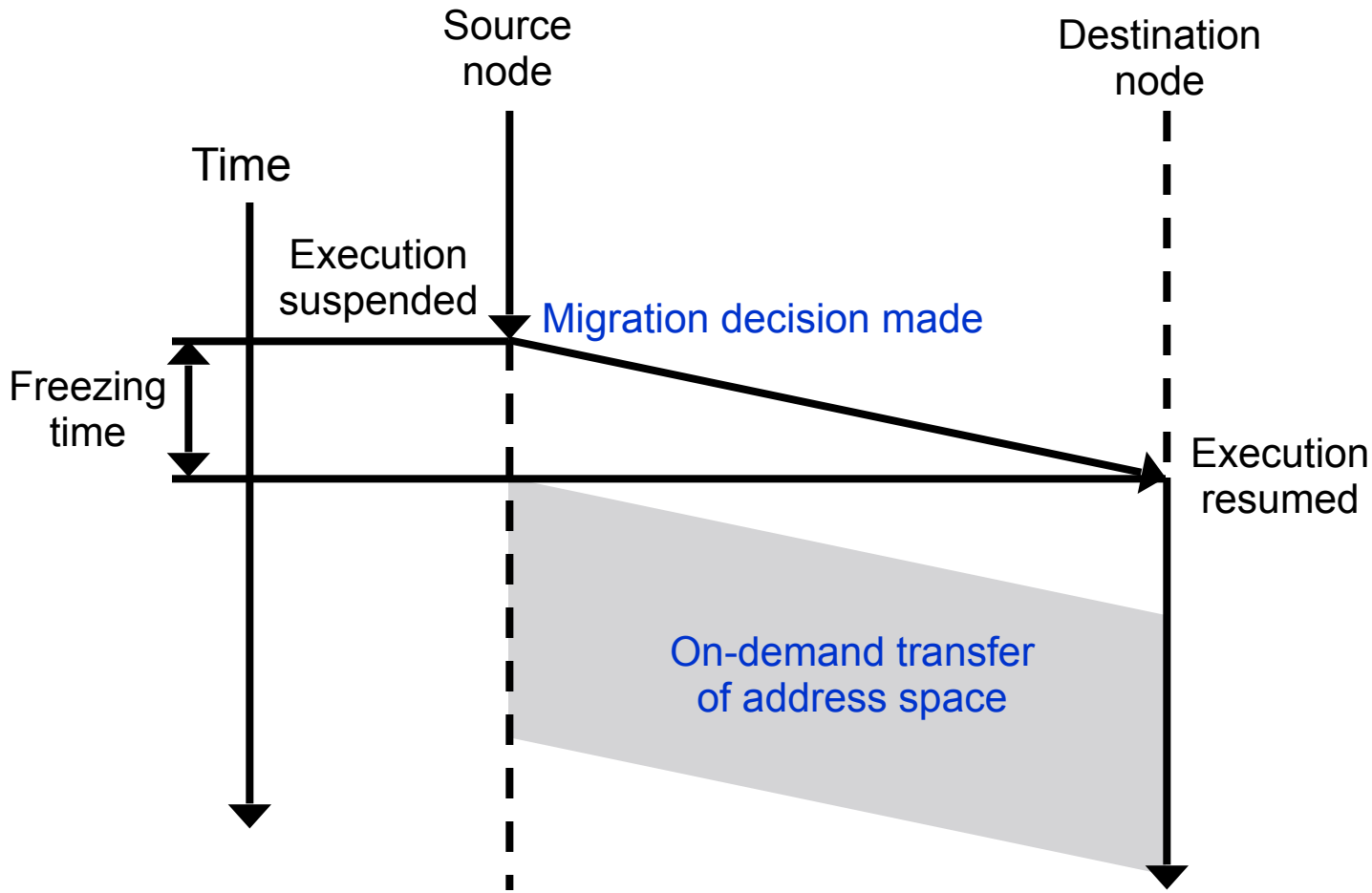
Address Space Transfer Mechanisms

- Pretransferring



Address Space Transfer Mechanisms

- Transfer on Reference



How to select the address space migration mechanism?

	Interactive Systems	Non-interactive Systems
High Network Bandwidth		
Low Network Bandwidth		

How to select the address space migration mechanism?

	Interactive Systems	Non-interactive Systems
High Network Bandwidth	Total Freezing	Transfer-on-Reference Pre-Transfer
Low Network Bandwidth		Transfer-on-Reference

How to select the address space migration mechanism?

	Interactive Systems	Non-interactive Systems
High Network Bandwidth	Total Freezing	Transfer-on-Reference Pre-Transfer
Low Network Bandwidth	Pre-transfer	Transfer-on-Reference

How to select the address space migration mechanism?

	Interactive Systems	Non-interactive Systems
High Network Bandwidth	Total Freezing	Transfer-on-Reference Pre-Transfer
Low Network Bandwidth	Pre-transfer	Transfer-on-Reference

Message-Forwarding Mechanisms

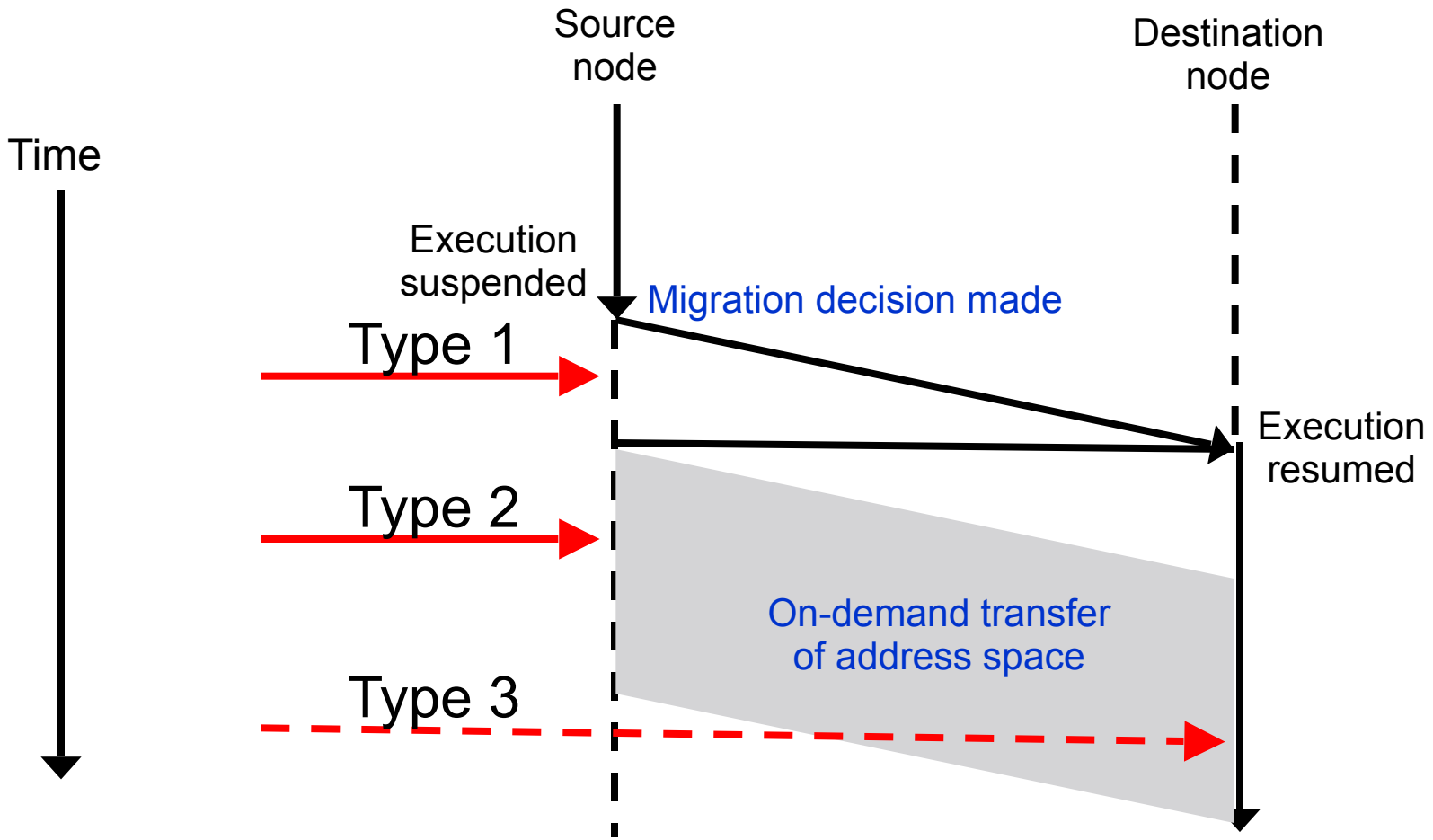
- In moving a process, it must be ensured that all pending, en-route, and future messages arrive at the process's new location.
- The messages to be forwarded can be classified into the following types:
  1. Messages received at the source node after the process's execution has been stopped on its source node and the process's execution has not yet been started on its destination node.
  2. Messages received at the source node after the process's execution has started on its destination node.
  3. Messages that are to be sent to the migrant process from any other node after it has started executing on the destination node.

Message-Forwarding Mechanisms

Messages should be forwarded.

Message-Forwarding Mechanisms

Messages should be forwarded.



Discussion: How will you design the message forwarding mechanism?

Message-Forwarding Mechanisms

Mechanism of Resending the Message

- Messages of types 1 and 2 are returned to the sender as not deliverable or are simply dropped, with the assurance that the sender of the message is storing a copy of the data and is prepared to retransmit it.
- In this method, upon receipt of a negative reply, the sender does a *locate* operation to find the new whereabouts of the process, and communication is reestablished.
- Messages of type 3 are sent directly to the process's destination node.
- The main drawback is that the mechanism is

Message-Forwarding Mechanisms

Mechanism of Resending the Message

- Messages of types 1 and 2 are returned to the sender as not deliverable or are simply dropped, with the assurance that the sender of the message is storing a copy of the data and is prepared to retransmit it.
- In this method, upon receipt of a negative reply, the sender does a *locate* operation to find the new whereabouts of the process, and communication is reestablished.
- Messages of type 3 are sent directly to the process's destination node.
- The main drawback is that the mechanism is

nontransparent to the processes interacting with the migrant process.

Message-Forwarding Mechanisms

Origin Site Mechanism

- The process identifier has the process's origin site embedded in it, each site is responsible for keeping information about the current locations of all the processes created on it.
- Messages for a particular process are always first sent to its origin site. The origin site then forwards the message to the process's current location.
- Drawbacks:

Message-Forwarding Mechanisms

Origin Site Mechanism

- The process identifier has the process's origin site embedded in it, each site is responsible for keeping information about the current locations of all the processes created on it.
- Messages for a particular process are always first sent to its origin site. The origin site then forwards the message to the process's current location.
- Drawbacks:
  - The method is not good from a reliability point of view because the failure of the origin site will disrupt the mechanism.
  - Another drawback is that there is a continuous load on the migrant process's origin site.

Message-Forwarding Mechanisms

Link Traversal Mechanism

- To redirect the messages of type 1, a message queue for the migrant process is created on its source node.
  - All messages in the queue are sent to the destination node as a part of the migration procedure.
- To redirect the messages of types 2 and 3, a forwarding address known as *link* is left at the source node pointing to the destination node of the migrant process, which has two components:
  - the first one is a system-wide, unique, process identifier and
  - the second one is the last known location of the process.
- The drawbacks are

Message-Forwarding Mechanisms

Link Traversal Mechanism

- To redirect the messages of type 1, a message queue for the migrant process is created on its source node.
  - All messages in the queue are sent to the destination node as a part of the migration procedure.
- To redirect the messages of types 2 and 3, a forwarding address known as *link* is left at the source node pointing to the destination node of the migrant process, which has two components:
  - the first one is a system-wide, unique, process identifier and
  - the second one is the last known location of the process.
- The drawbacks are poor efficiency and without reliability.

Message-Forwarding Mechanisms

Link Update Mechanism

- During the transfer phase, the source node sends link-update messages. These link update messages tell the new address of each link held by the migrant process and are acknowledge for synchronization.
  - This task is not expensive since it is performed in parallel.
- Thus, messages sent to the migrant process on any of its links will be sent directly to the migrant process's new node.
- Messages of types 1 and 2 are forwarded to the destination node by the source node and messages of type 3 are sent directly to the process's destination node.

Comparison

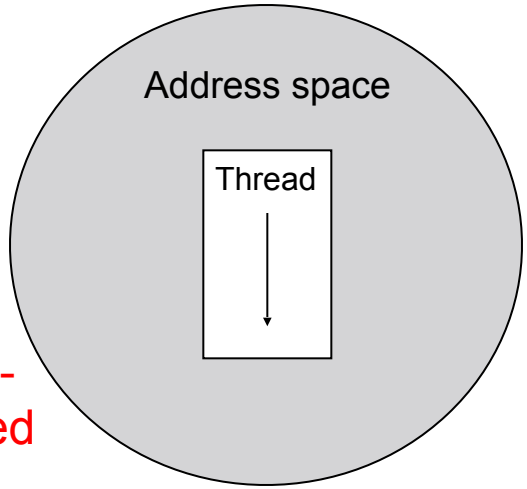
- Mechanisms:
  - Msg Resending
  - Origin Site
  - Link Traversal
  - Link Update
- Implementation Cost:
  - Lowest: Resending
  - Highest: Link Update
- Reliability:
  - Lowest: Link Traversal
  - Highest: Msg Resending
- Efficiency:
  - Least: Link Traversal (network)/Origin site (extra workload)
  - Most: Link Update

Advantages of Process Migration

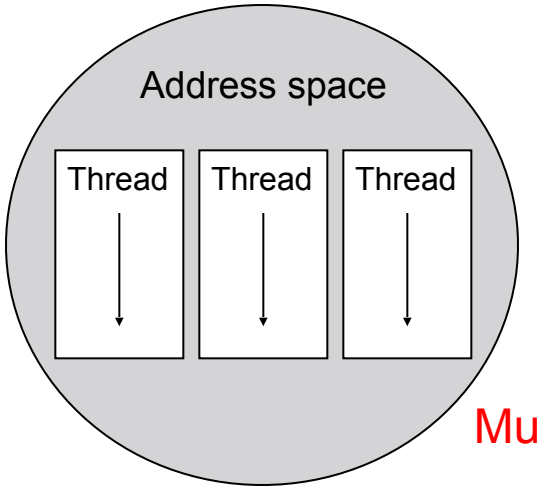
- Process migration facility may be implemented in a distributed system for providing:
  - Reducing average response time of processes
  - Speeding up individual jobs
  - Gaining higher throughput
  - Utilizing resources effectively
  - Reducing network traffic
  - Improving system reliability
  - Improving system security

Threads

- Threads are a popular way to improve application performance through parallelism.
  - In traditional OSs the basic unit of CPU utilization is a process, which has its own program counter, register states, stack, and address space.
  - In OSs with threads facility, a process consists of an address space and one or more threads of control.
  - Each thread of a process has its own program counter, register states, and stack. But all the threads of a process share the same address space.



Single-threaded



Multithreaded

Threads (Cont.)

- In addition, all threads of a process also share the same set of operating system resources, such as
  - open files,
  - child processes,
  - semaphores,
  - signals,
  - accounting information,
  - and so on.
- There is no protection between the threads of a process.
- Protection between different processes is needed because different processes may belong to different users.

Threads (Cont.)

- Threads share a CPU in the same way as processes do.
- On a uniprocessor, threads run in quasi-parallel (time sharing), whereas on a shared-memory multiprocessor, as many threads can run simultaneously as there are processors.
- Threads can
  - create child threads,
  - block waiting for system calls to complete,
  - change states during their course of execution.
- At a particular instance of time, a thread can be in any one of several states: running, blocked, ready, or terminated.
- Threads are often referred to as *lightweight processes* and traditional processes are referred to as *heavyweight processes.*

Motivations for Using Threads

- The overheads involved in creating a new process are in general considerably greater than those of creating a new thread within a process.
- Switching between threads sharing the same address space is considerably cheaper than switching between processes that have their own address spaces.
- Threads allow parallelism to be combined with sequential execution and blocking system calls. Parallelism improves performance and blocking system calls make programming easier.
- Resource sharing can be achieved more efficiently and naturally between threads of a process than between processes because all threads of a process share the same address space.

Class Discussion

- Suppose you are implementing a file server which accepts the requests from other clients. How would you design the file server to achieve maximum throughput?
- Please consider: implementation overhead, average response time, and run-time overhead.

Construction Models of a System Process

- As a single-thread process
  - This model uses blocking system calls but without any parallelism.
  - The programming of the server process is simple, however, the performance is often unacceptable.
  - If it is used for the file sever, the CPU remains idle while the file server is waiting for a reply from the disk server.
- As a finite-state machine
  - This model supports parallelism but with nonblocking system calls.
  - An event queue is maintained instead of blocking.
  - It is difficult to program due to the use of nonblocking system calls.
- As a group of threads
  - This model supports parallelism with blocking system calls.
  - The server process is comprised of a single *dispatcher* thread and multiple *worker* threads, which are either be created dynamically or a pool of pre-created threads.
  - It has good performance and is also easy to program.

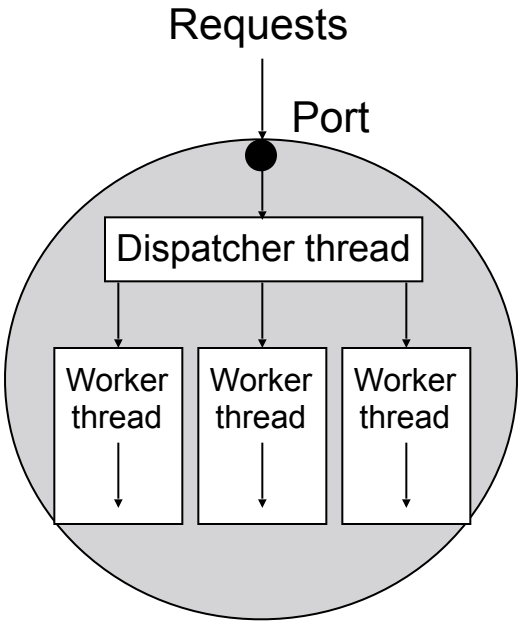
Models for Organizing Threads

- Depending on an application's needs, the threads of a process of the application can be organized in different ways.

- Dispatcher-workers model
- Team model
- Pipeline model

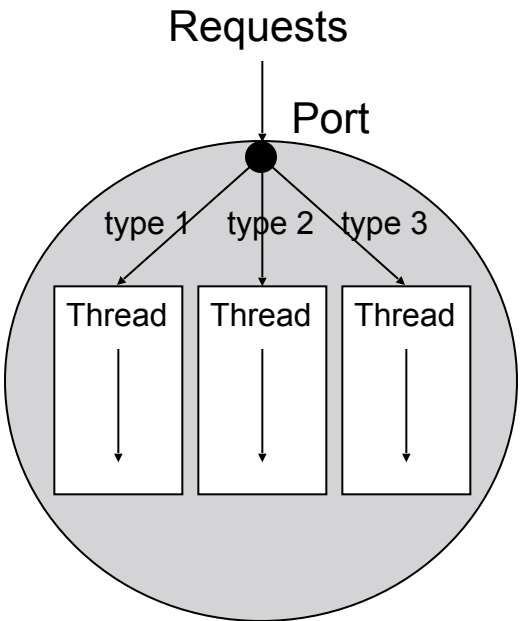
Models for Organizing Threads (Cont.)

- Dispatcher-Workers Model
  - The process consists of a single dispatcher thread and multiple worker threads.
  - The dispatcher thread accepts requests from clients and dispatches the request to one of the free worker threads for further processing of the request.



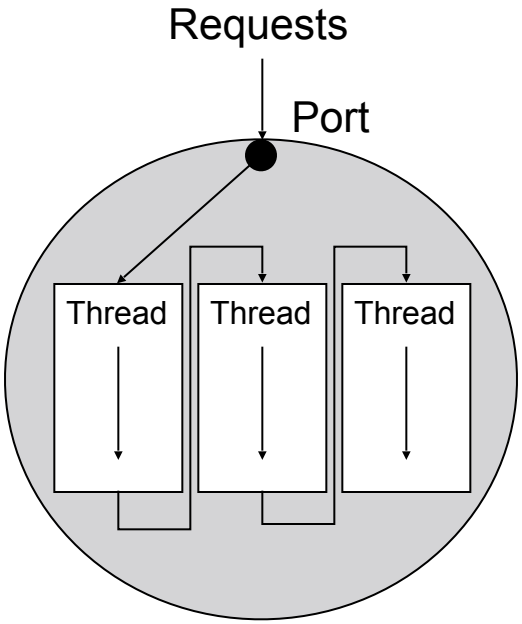
Models for Organizing Threads (Cont.)

- Team Model
  - All threads behave as equals in the sense that there is no dispatcher-worker relationship for processing clients' requests.
  - This model is often used for implementing specialized threads within a process.
  - Each thread is specialized in servicing a specific type of request.
  - Multiple types or requests can be simultaneously handled by the process.



Models for Organizing Threads (Cont.)

- Pipeline Model
  - It is useful for applications based on the producer-consumer model.
  - The threads of a process are organized as a pipeline so that the output data generated by the first thread is used for processing by the second thread, and so on.
  - The output of the last thread in the pipeline is the final output of the process to which the threads belong.



Issues in Designing a Thread Package

- A system that supports threads facility must provide a set of primitives to its users for threads-related operations.
  - These primitives of the system are said to form a *thread package*.
- Some of the important issues are:
  - Threads Creation
  - Threads Termination
  - Threads Synchronization
  - Threads Scheduling

Issues in Designing a Thread Package (Cont.)

- Thread Creation
  - Threads can be created either statically or dynamically.
  - In the dynamic approach, a process is started with a single thread, new threads are created as and when needed during the execution of the process.
    - A thread may destroy itself when it finishes its job by making an exit call.
  - In the static approach, a fixed stack is allocated to each thread.
  - The system call returns a thread identifier for the newly created thread.

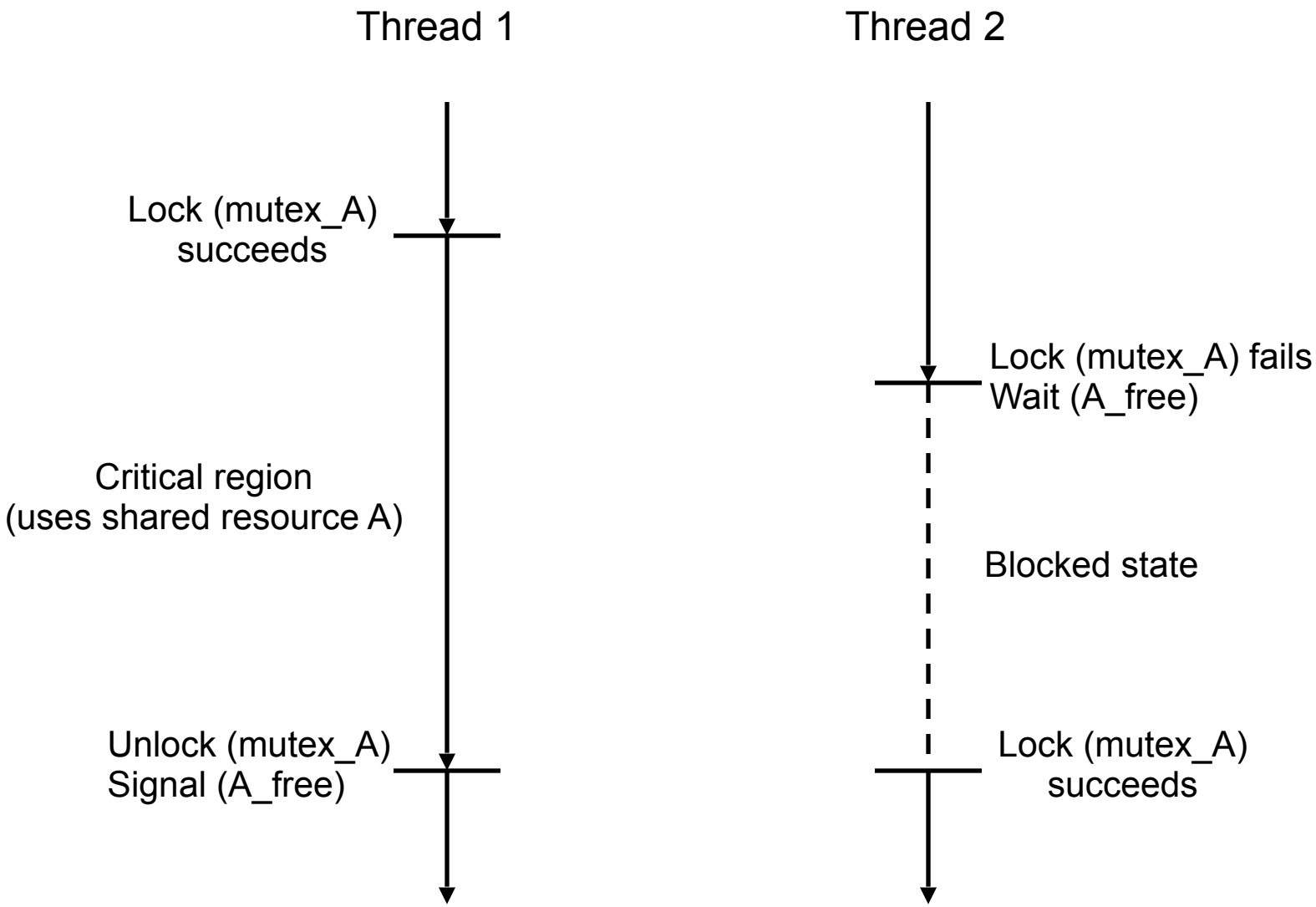
Issues in Designing a Thread Package (Cont.)

- Thread Termination
  - Termination of threads is performed in a manner similar to the termination of conventional processes.
    - A thread may either destroy itself when it finishes its job by making an exit call or be killed from outside by using the kill command and specifying the thread identifier as its parameter.
  - In many cases, threads are never terminated.
    - In a process that uses statically created threads, the number of threads remains constant for the entire life of the process.
    - All its threads are created immediately after the process starts up and then these threads are never killed until the process terminates.

Issues in Designing a Thread Package (Cont.)

- Thread Synchronization
  - Some mechanism must be used to prevent multiple threads from trying to access the same data simultaneously.
  - The execution of *critical regions* in which the same data is accessed by the threads must be *mutually exclusive*.
  - Two commonly used mutual exclusion techniques in a threads package are:
    - Mutex variable
      - It is like a binary semaphore that is always in one of two states, locked or unlocked.
    - Condition variable
      - It is associated with a mutex variable and reflects a Boolean state of that variable.

Issues in Designing a Thread Package (Cont.)



Issues in Designing a Thread Package (Cont.)

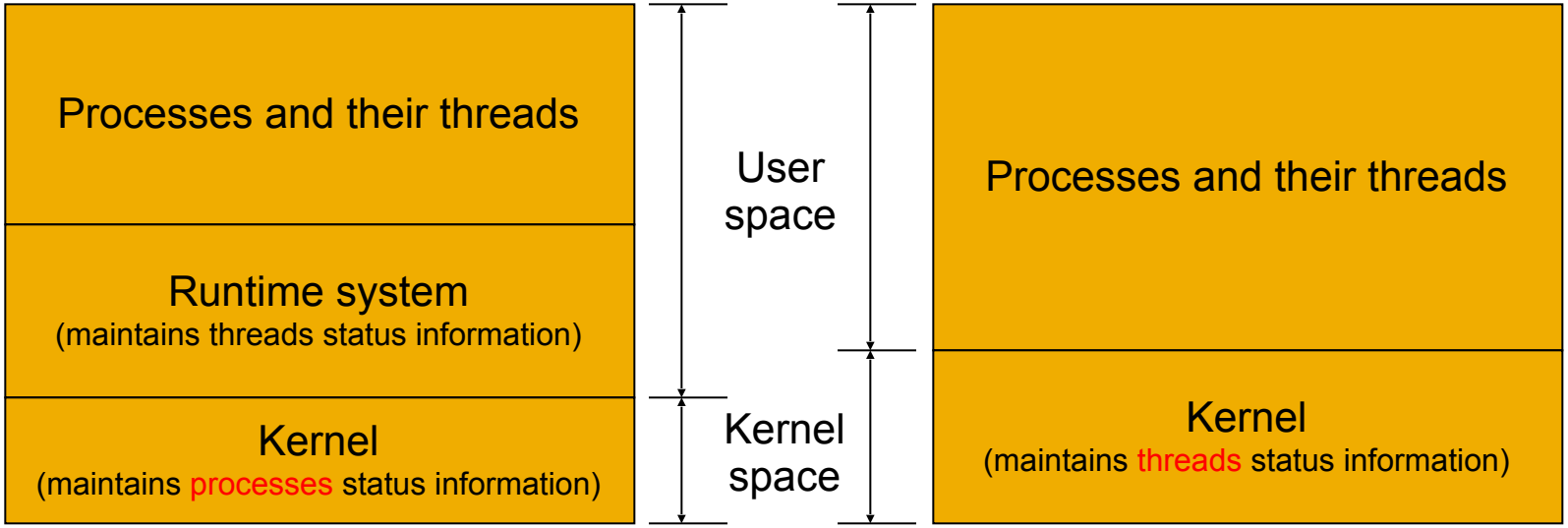
- Thread Scheduling
  - Threads packages often provide calls to give the users the flexibility to specify the scheduling policy to be used for their applications.
  - Some of the special features for threads scheduling
    - Priority assignment facility
      - FIFO vs. Round-Robin
      - Preemptive vs. Non-preemptive
    - Flexibility to vary quantum size dynamically
    - Handoff scheduling
      - It allows a thread to name its successor.
    - Affinity scheduling
      - A thread is scheduled if that part of its address space is still in the CPU's cache.

Issues in Designing a Thread Package (Cont.)

- Signal Handling
  - Signals provide software-generated interrupts and exceptions.
    - A signal must be handled properly no matter which thread of the process receives it.
    - Signals must be prevented from getting lost.
      - This happens because an exception condition causing the signal is stored in a process-wide global variable that is overwritten by another exception condition.
  - An approach for handling the former issue is to create a separate exception handler thread in each process.
  - An approach for handling the latter issue is to assign each thread its own private global variables for signaling exception conditions.

Implementing a Threads Package

- A thread package can be implemented either in user space or in the kernel.
  - The two approaches are referred to as *user-level* and *kernel-level*.



Implementing a Threads Package (Cont.)

- Comparisons of user-level and kernel-level approaches.
  - Low implementation overhead and flexible design for user-level thread.
  - Autonomously scheduling policies for threads.
  - Low context switch overhead between the threads in a process for user-level thread.
  - Better scalability (or easier to maintain) for user-level thread.
  - However,
    - It is difficult to implement quantum-based round-robin scheduling policy for user-level threads. Why?
    - Blocking system calls have to be avoided on user-level thread to avoid context switches on process-level.

Case Study: DCE Threads

- IEEE has drafted a POSIX threads standard known as *P-Threads*.
- The DCE Threads are developed by the OSF.
  - Which is based on the P1003.4a POSIX standard.
  - The user-level approach is used for implementing.
  - DCE provides a set of user-level library procedures. To access thread services from applications written in C, DCE specifies an API that is compatible to the POSIX standard.
  - If a system supporting DCE has operating system kernel support for threads, the DCE is set up to use this facility.

Case Study: DCE Threads (Cont.)

- Threads Management
  - *pthread_create* is used to create a new thread.
  - *pthread_exit* is used to terminate the calling thread.
  - *pthread_join* is used to cause the calling thread to block itself until the thread specified in this routine's argument terminates.
  - *pthread_detach* is used by a parent thread to disown a child thread.
  - *pthread_cancel* is used by a thread to kill another thread.
  - *pthread_setcancel* is used by a thread to enable or disable ability of other threads to kill it.

Case Study: DCE Threads (Cont.)

- DCE supports the following types of mutex variables:
  - Fast.
    - A fast mutex variable is one that causes a thread to block when the thread attempts to lock an already locked mutex variable.
  - Recursive.
    - It allows a thread to lock an already locked mutex variable, i.e., nested locking of a recursive mutex variable is permitted. It will never lead to deadlock.
  - Norecursive.
    - It neither allows a thread to lock an already locked mutex variable nor causes the thread to block.
    - Rather, an error is returned.

Case Study: DCE Threads (Cont.)

- Thread Scheduling
  - A user can choose from one of the following threads-scheduling algorithms:
    - First in, first out (FIFO)
    - Round robin (RR)
    - Default

Case Study: DCE Threads (Cont.)

- Signal Handling
  - An exception condition is handled by the thread in which it occurs.
  - However, an external interrupt is handled by all the concerned threads.
- Error Handling
  - P-Threads calls report errors by setting a global variable, *errno*, and returning -1.