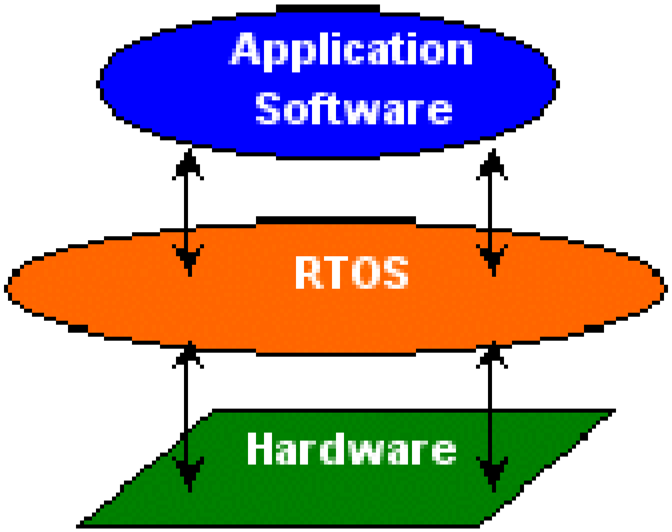
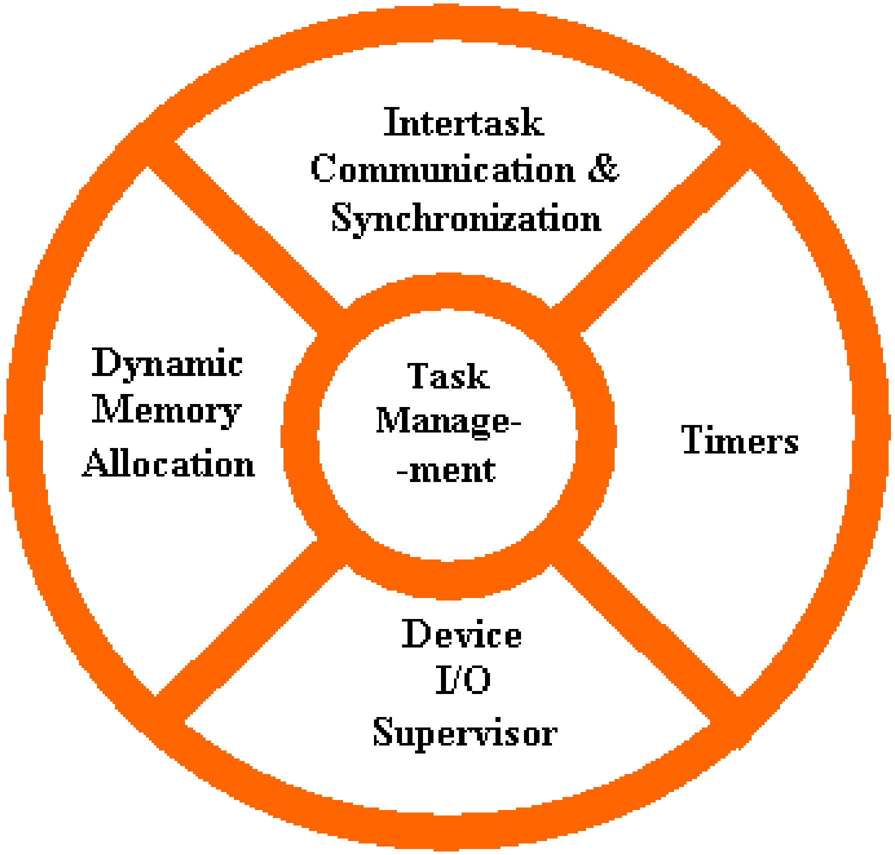
Real-Time Operating System

ELC 4438 – Spring 2016

Liang Dong
Baylor University

RTOS – Basic Kernel Services





Task Management

- Scheduling is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communication bandwidth).

- The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (executing more than one process at a time) and multiplexing (transmit multiple data streams simultaneously across a single physical channel).

Task Management

- Polled loops; Synchronized polled loops
- Cyclic Executives (round-robin)
- State-driven and co-routines
- Interrupt-driven systems
  - Interrupt service routines
  - Context switching

Interrupt-driven Systems

```
void main(void)
{
    init();
    while(true);
}

void int1(void)
{
    save(context);
    task1();
    restore(context);
}

void int2(void)
{
    save(context);
    task2();
    restore(context);
}
```

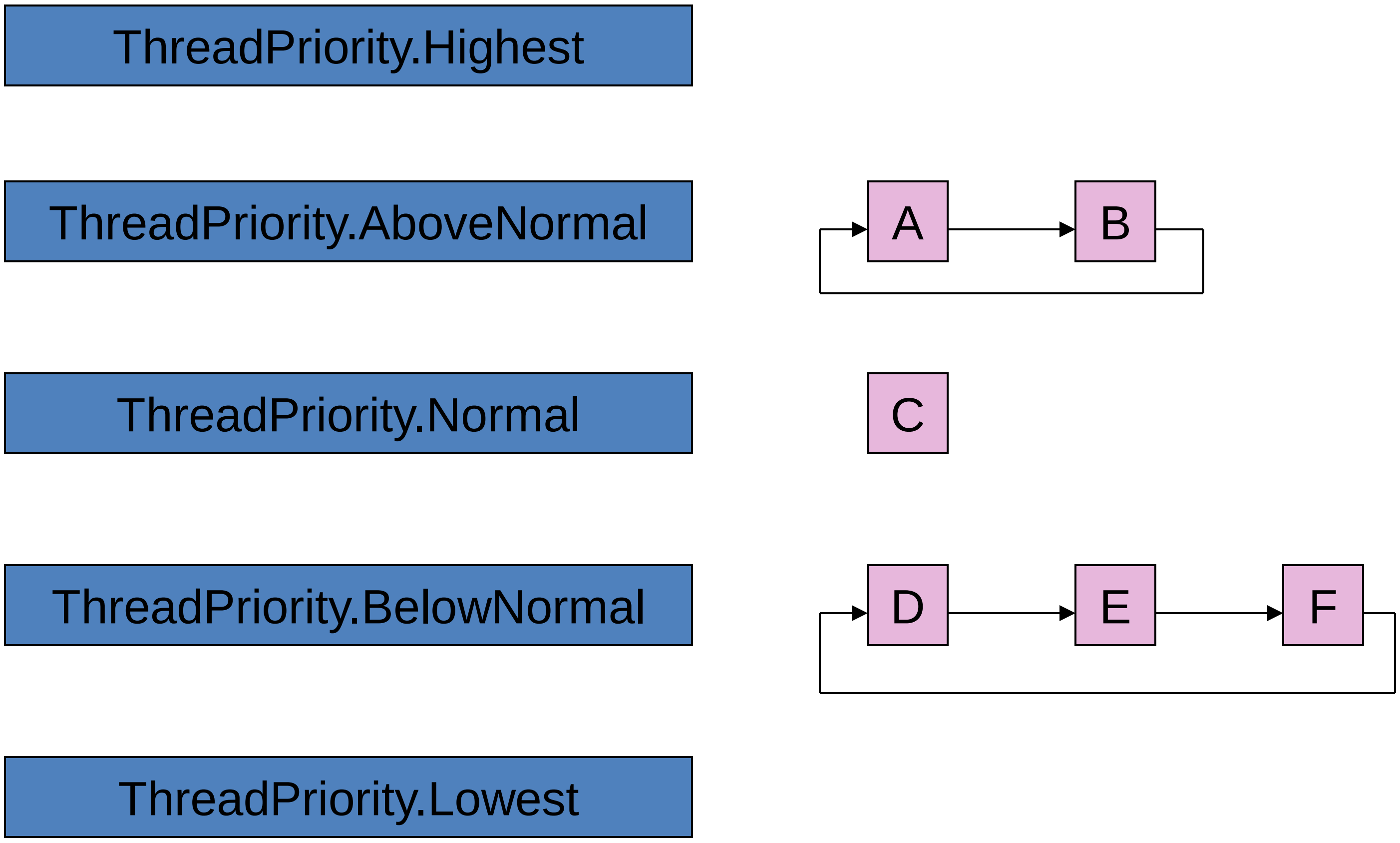
Task scheduling

- Most RTOSs do their scheduling of tasks using a scheme called "priority-based preemptive scheduling."
- Each task in a software application must be assigned a priority, with higher priority values representing the need for quicker responsiveness.
- Very quick responsiveness is made possible by the "preemptive" nature of the task scheduling. "Preemptive" means that the scheduler is allowed to stop any task at any point in its execution, if it determines that another task needs to run immediately.

Hybrid Systems

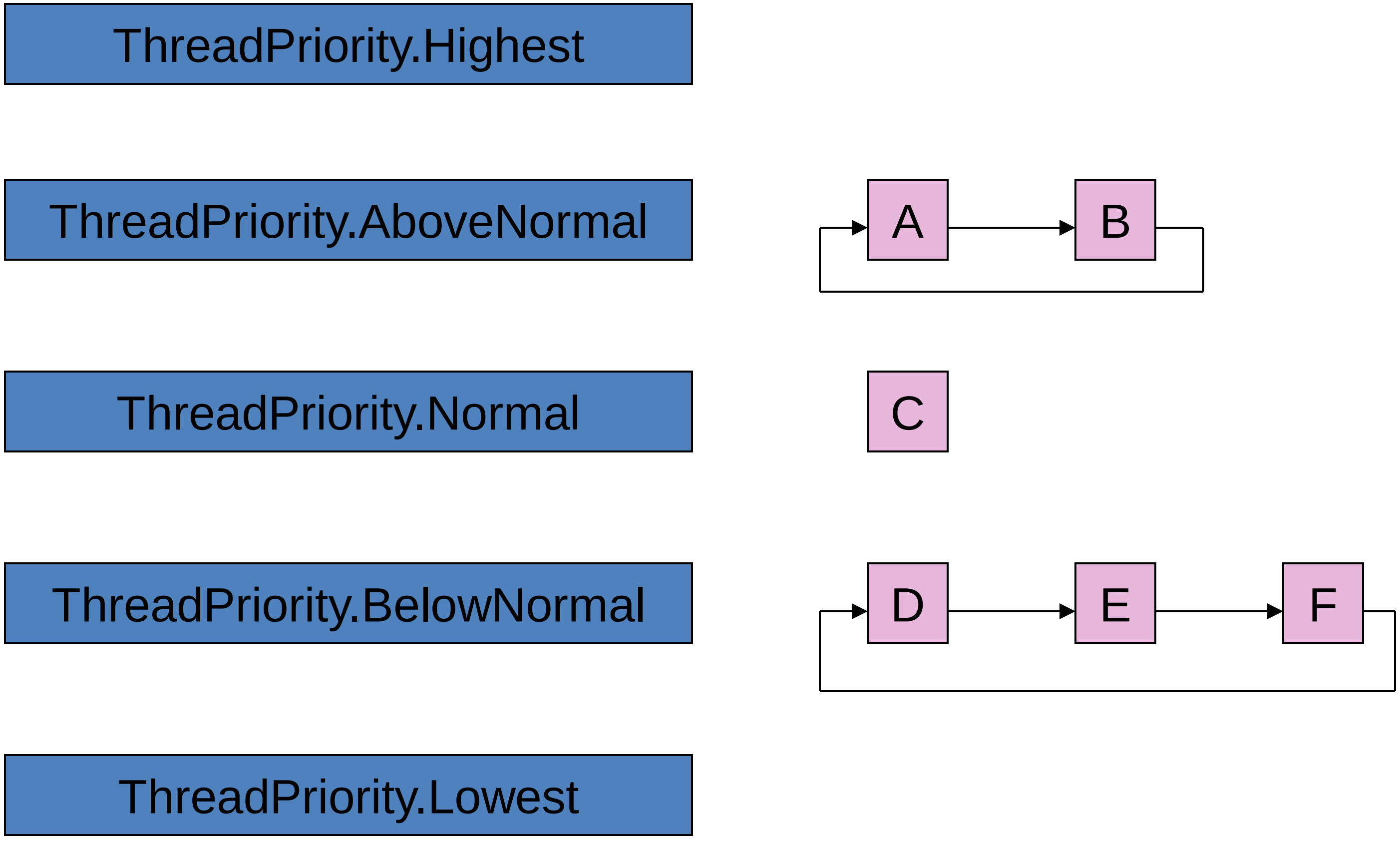
- A hybrid system is a combination of round-robin and preemptive-priority systems.
  - Tasks of higher priority can preempt those of lower priority.
  - If two or more tasks of the same priority are ready to run simultaneously, they run in round-robin fashion.

Thread Scheduling



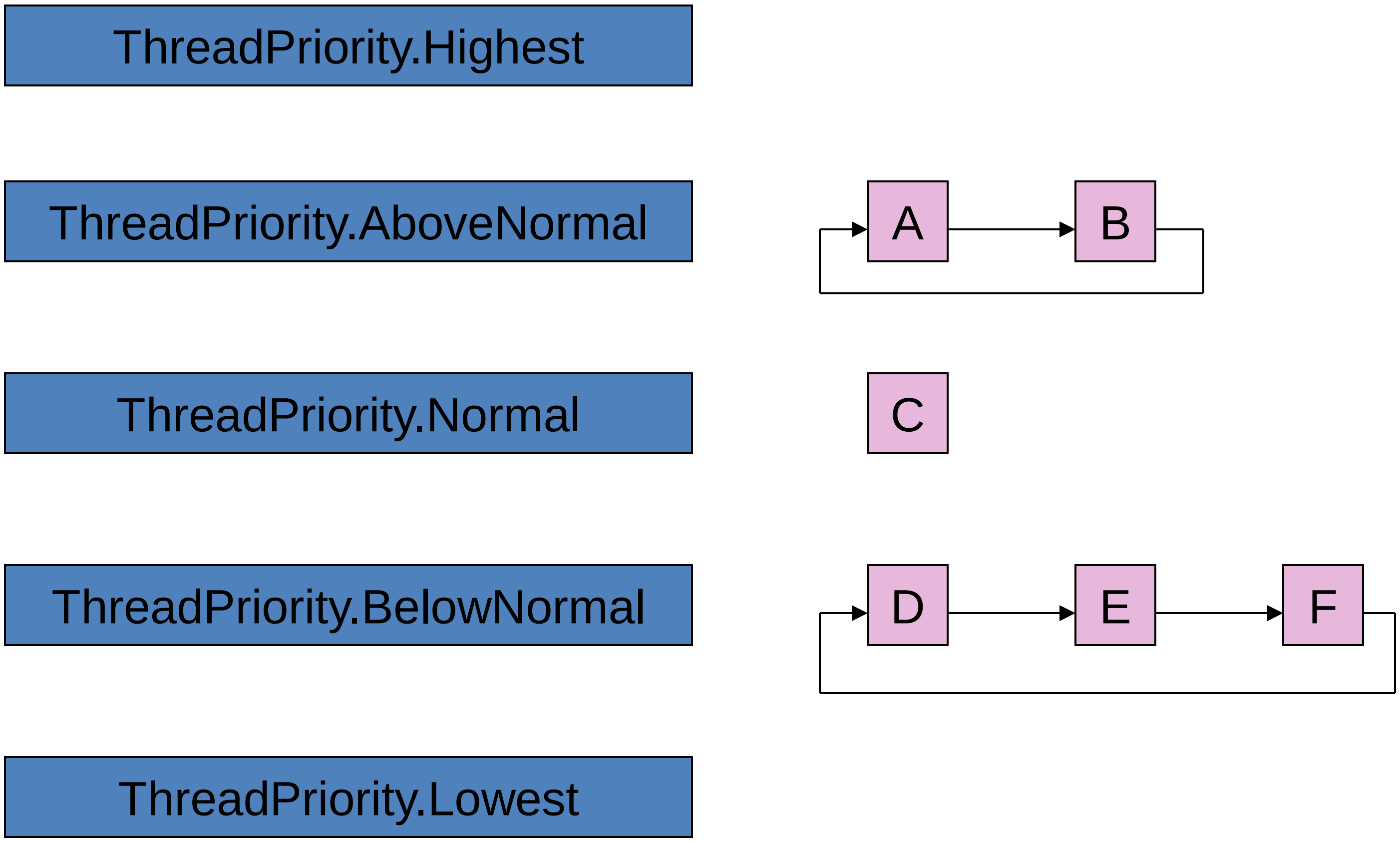
Default priority is Normal.

Thread Scheduling



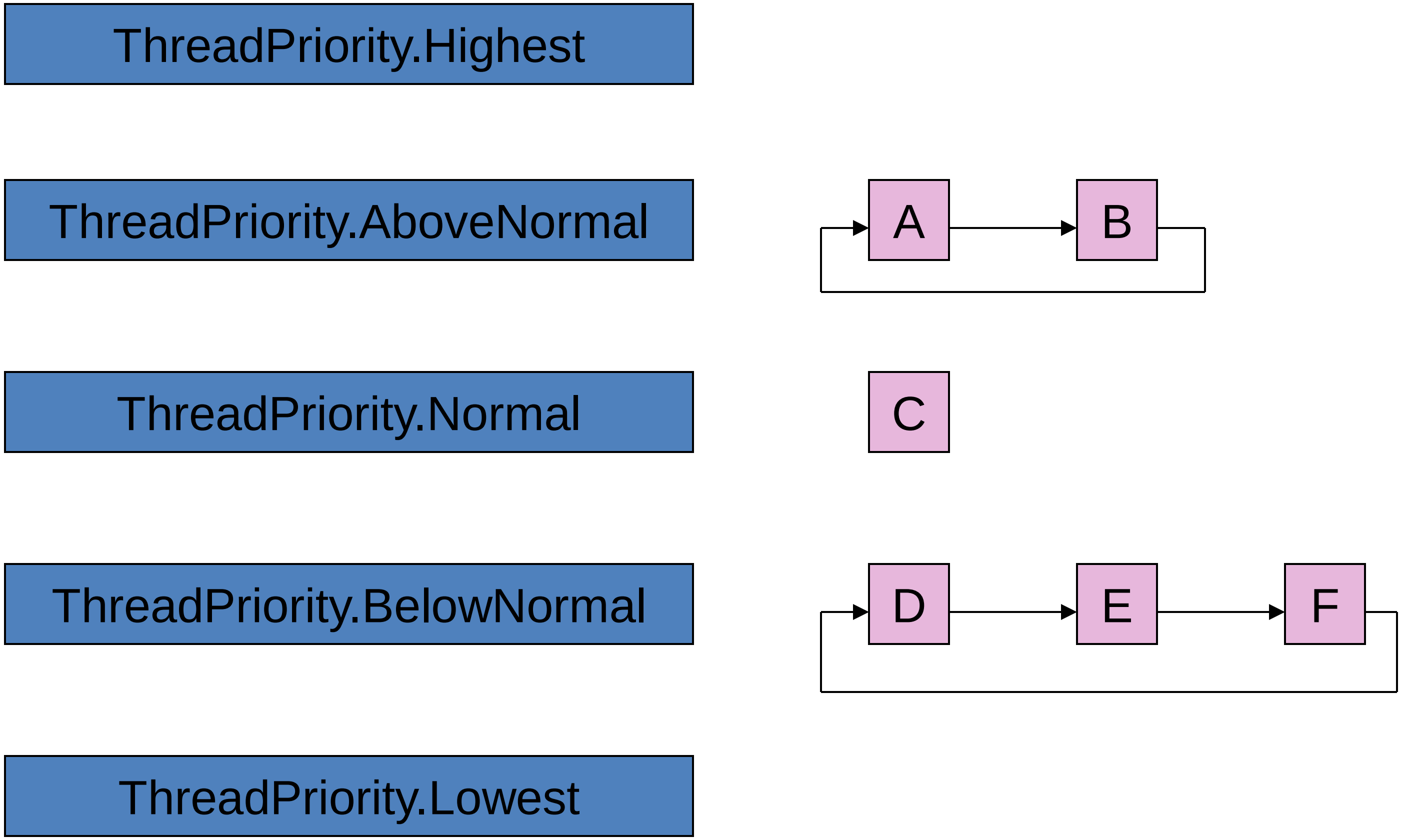
Threads A and B execute, each for a quantum, in round-robin fashion until both threads complete.

Thread Scheduling



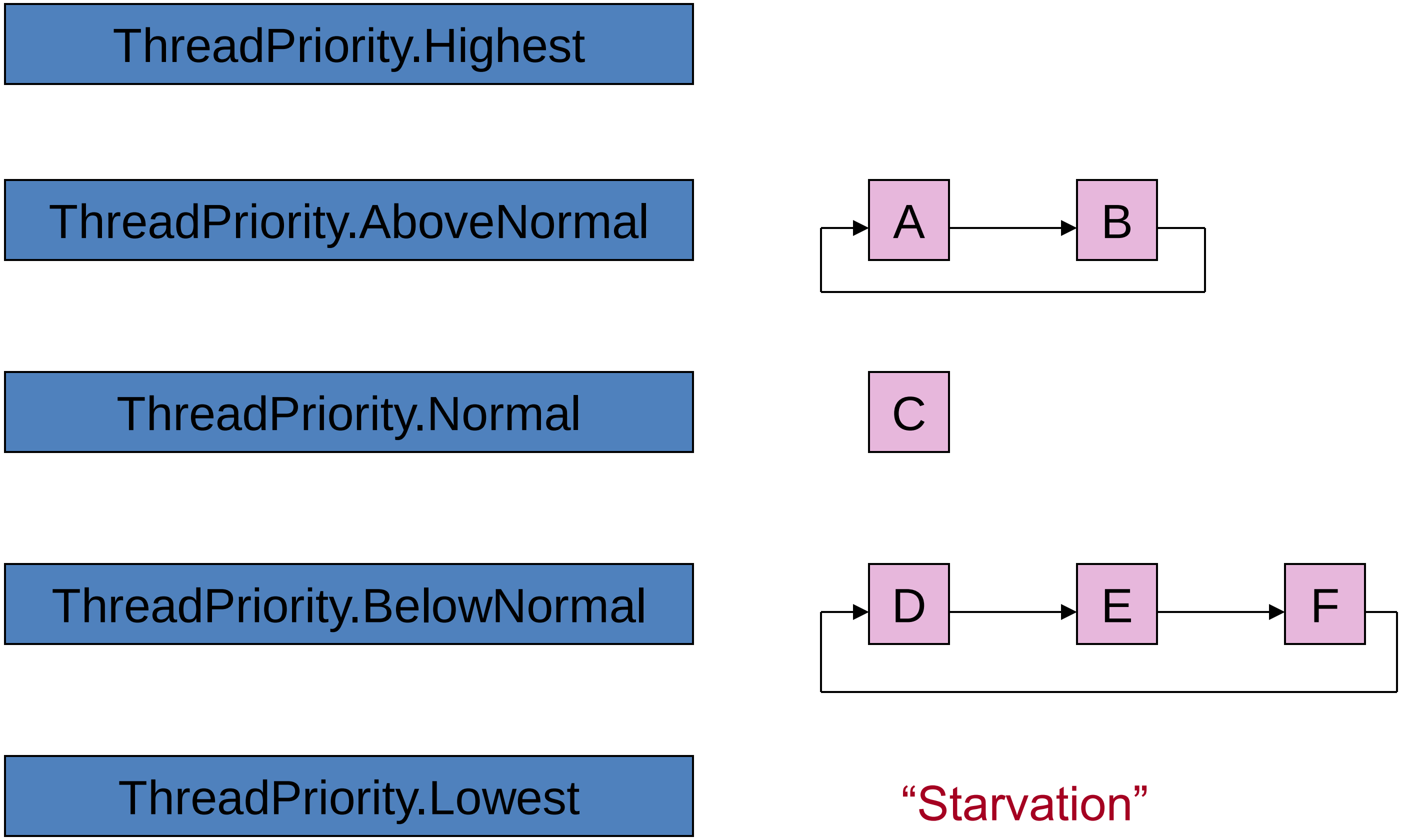
Then thread C runs to completion.

Thread Scheduling



Next threads D, E, F execute in round-robin fashion until they all complete execution.

Thread Scheduling



Foreground/Background Systems

- A set of interrupt-driven or real-time processes called the foreground and a collection of noninterrupt-driven processes called the background.
- The foreground tasks run in round-robin, preemptive priority, or hybrid fashion.
- The background task is fully preemptable by any foreground task and, in a sense, represents the lowest priority task in the system.

Foreground/Background Systems

- All real-time solutions are just special cases of the foreground/background systems.
- The polled loop is simply a foreground/background system with no foreground, and a polled loop as a background.
- Interrupt-only systems are foreground/background systems without background processing.

RTOSs vs. general-purpose operating systems

- Many non-real-time operating systems also provide similar kernel services. The key difference between general-computing operating systems and real-time operating systems is the need for " **deterministic** " timing behavior in the real-time operating systems.

- Formally, "deterministic" timing means that operating system services consume only known and expected amounts of time.

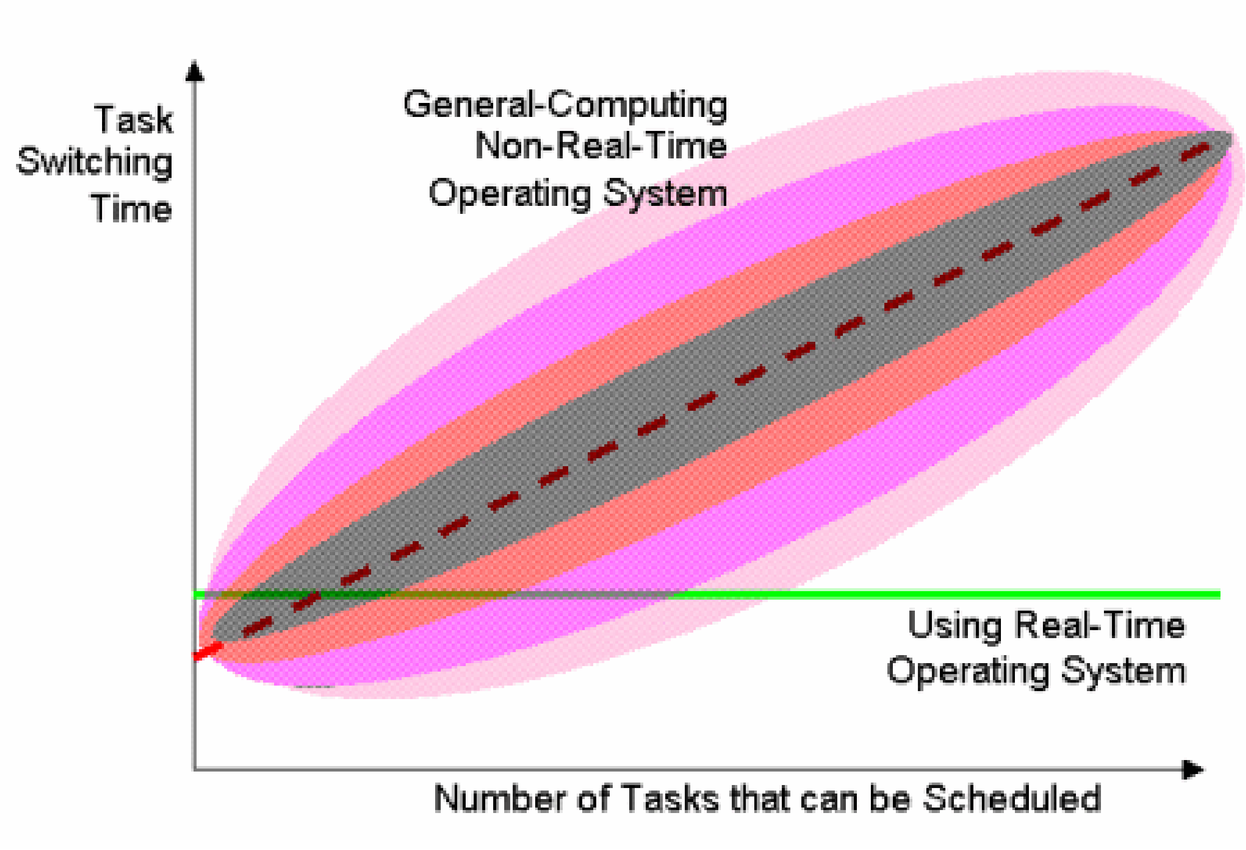
- In theory, these service times could be expressed as mathematical formulas. These formulas must be strictly algebraic and not include any random timing components.

RTOSs vs. general-purpose operating systems

- General-computing non-real-time operating systems are often quite non-deterministic. Their services can inject random delays into application software and thus cause slow responsiveness of an application at unexpected times.

- Deterministic timing behavior was simply not a design goal for these general-computing operating systems, such as Windows, Unix, Linux.

- On the other hand, real-time operating systems often go a step beyond basic determinism. For most kernel services, these operating systems offer constant **load-independent** timing.



The horizontal solid green line shows the task switching time characteristic of a real-time operating system. It is constant, independent of any load factor such as the number of tasks in a software system.

Intertask Communication & Sync

- Previously, we assume that all tasks are independent and that all tasks can be preempted at any point of their execution.
- In practice, task interaction is needed.
- The main concern is how to minimize blocking that may arise in a uniprocessor system when concurrent tasks use shared resources.

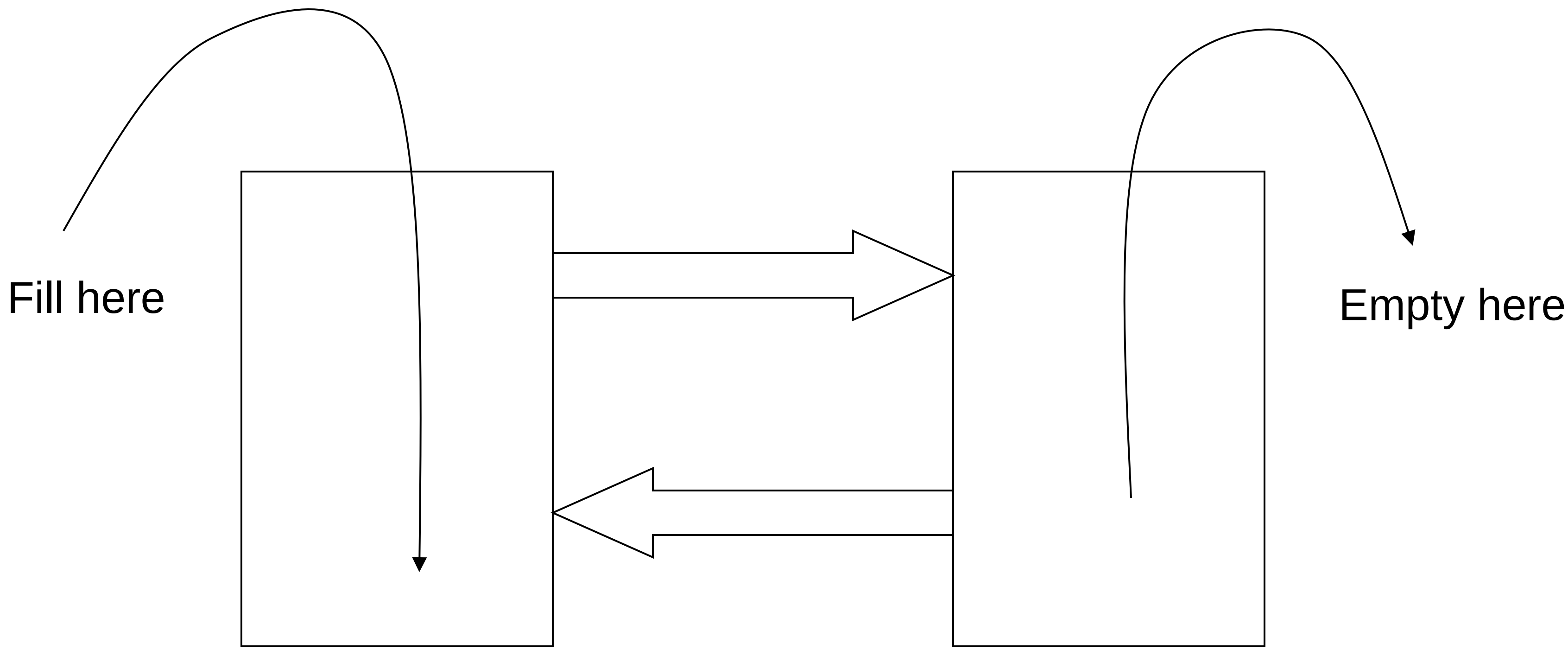
Buffering Data

- To pass data between tasks in a multitasking system, the simplest way is to use global variables.

- One of the problems related to using global variables is that tasks of higher- priority can preempt lower-priority routines at inopportune times, corrupting the global data.

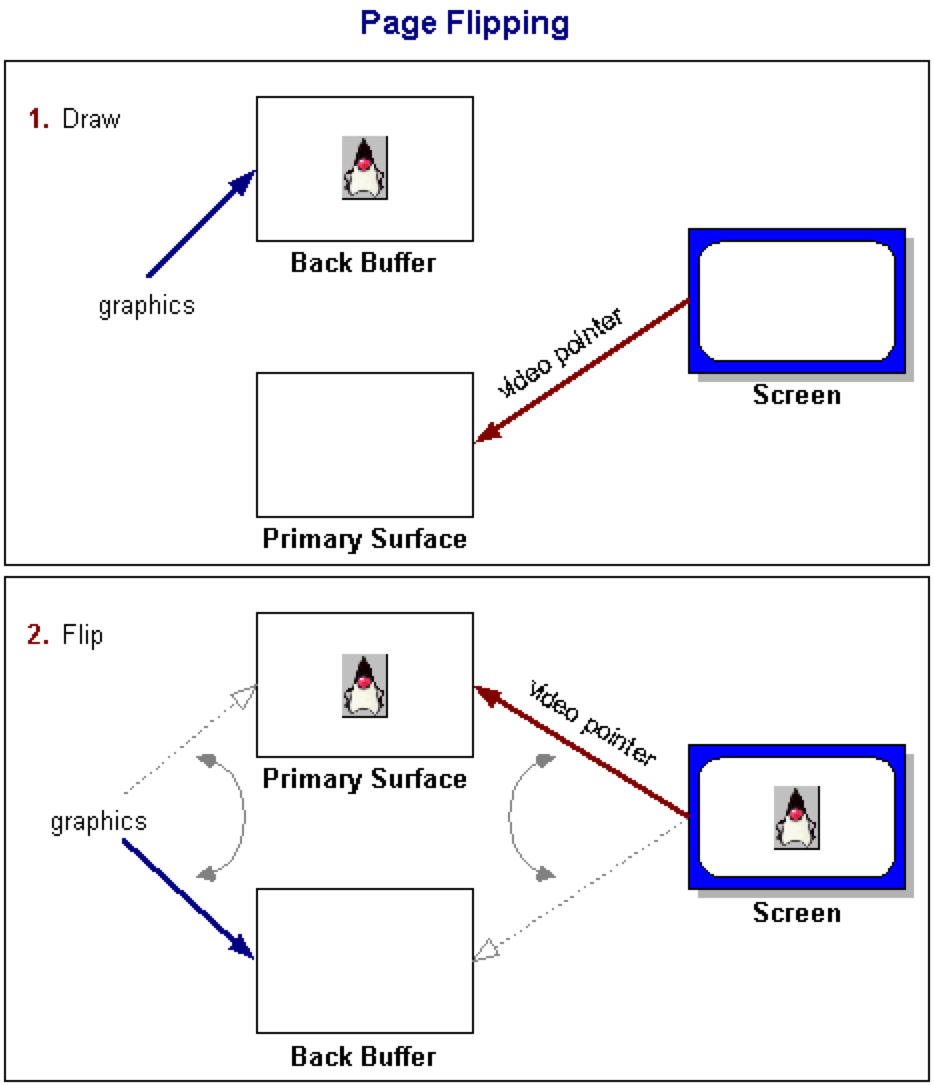
- Data buffer

Time-Relative Buffering



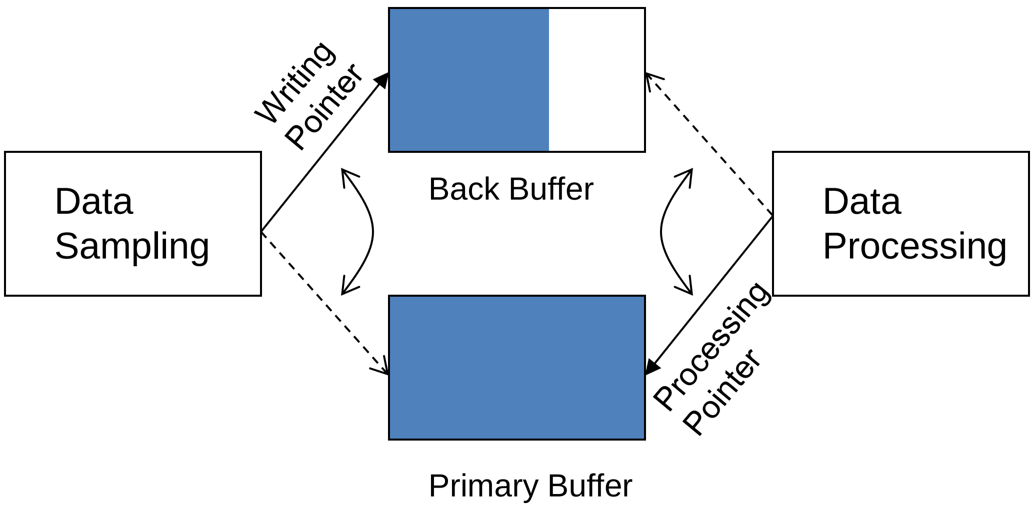
Swap buffers with interrupts off

Page Flipping via Pointer Switching



When a page flip occurs, the pointer to the old back buffer now points to the primary surface and the pointer to the old primary surface now points to the back buffer memory. This sets you up automatically for the next draw operation.

Receiving and Processing Buffers

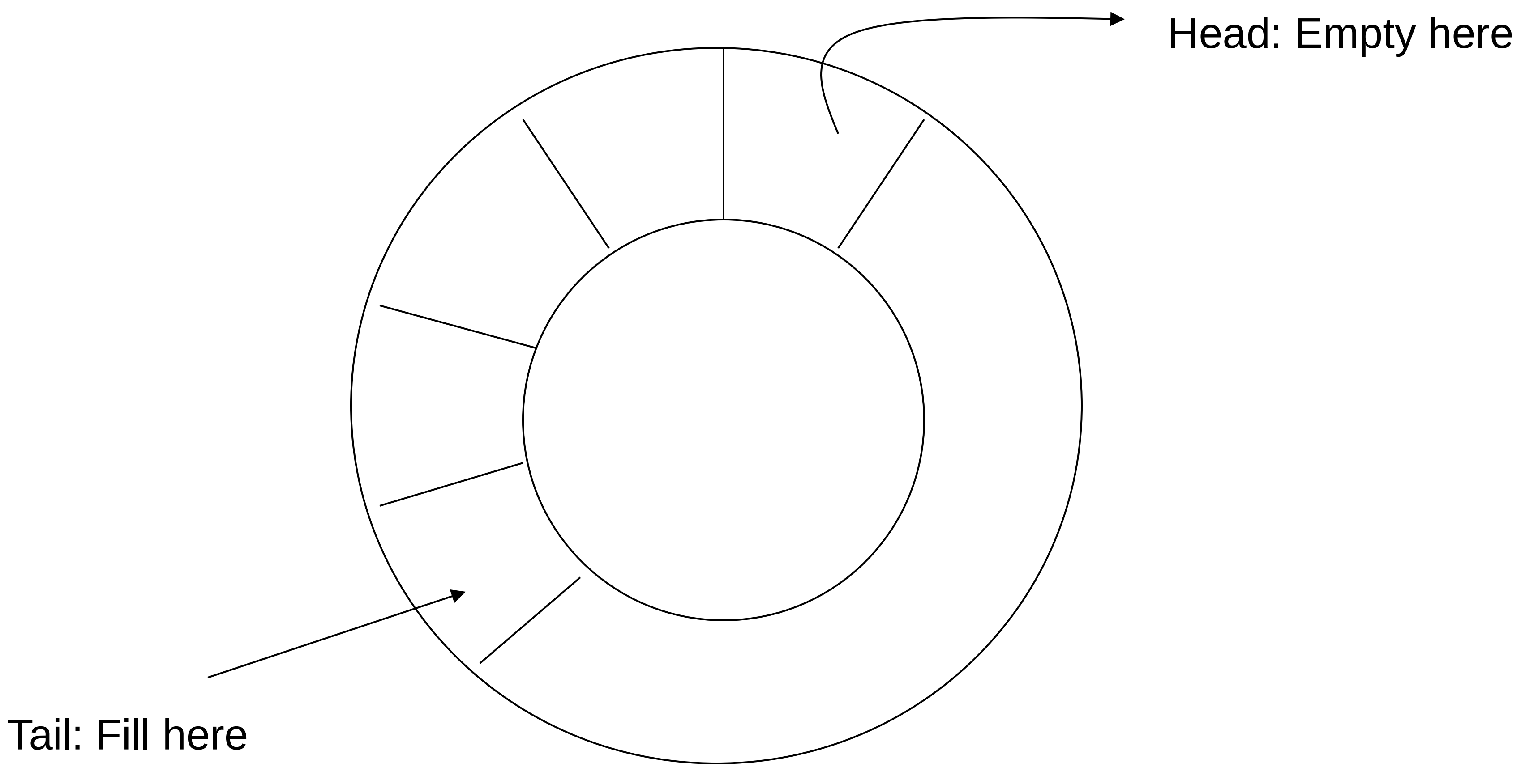


Double-Buffering for Data Reception and Process

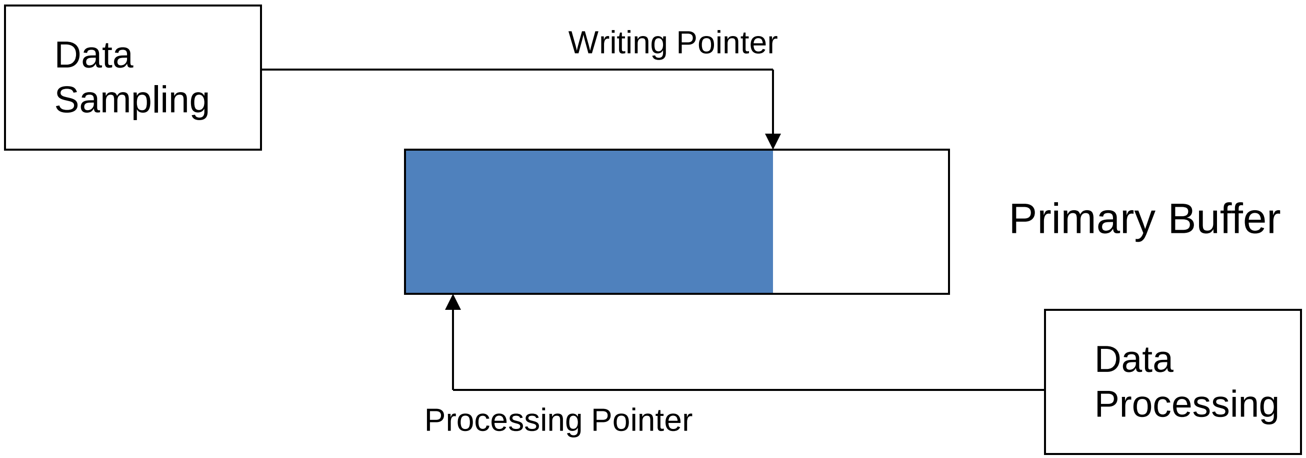
Time-Relative Buffering

- Double-buffering uses a hardware or software switch to alternate the buffers.
- Applications: disk controller, graphical interfaces, navigation equipment, robot controls, etc.

Circular Buffer



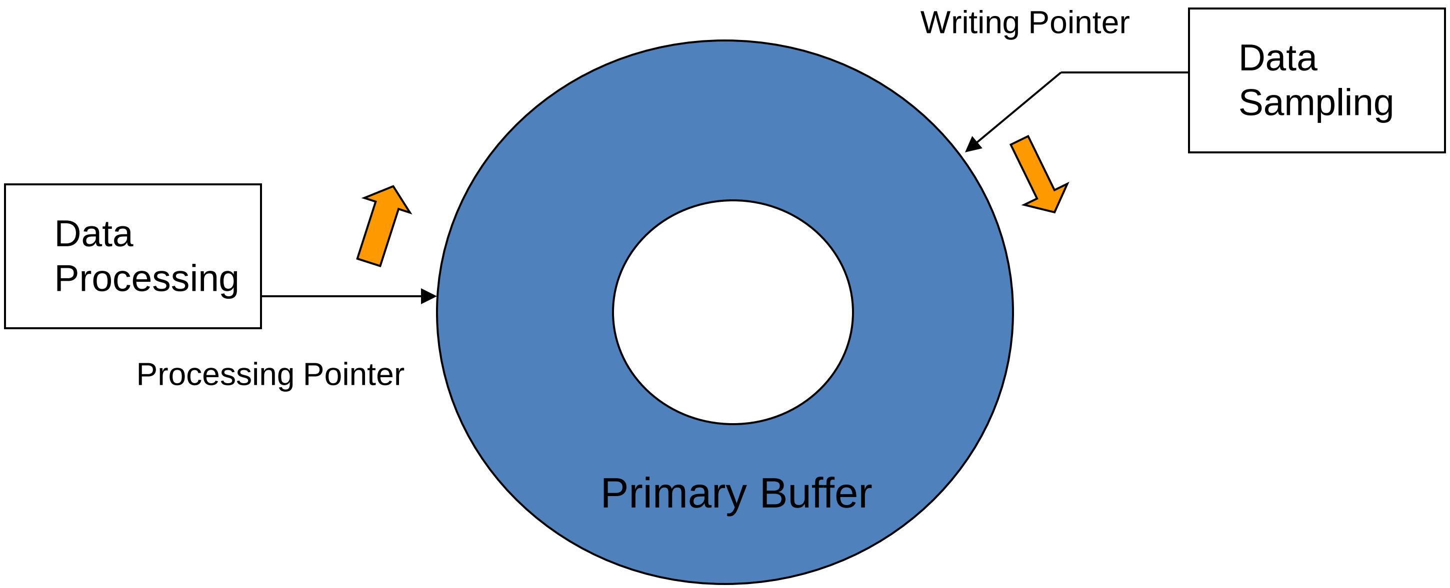
Circular Buffering



Circular-Buffering for Data Reception and Process

Writing_Pointer := mod (total_writing_count, buffer_size);
Processing_Pointer := mod(total_processing_count, buffer_size);

Circular Buffering (Cont.)



Pointers’ Chases in Circular-Buffering

Mailboxes

- Mailboxes or message exchanges are an intertask communication device available in many full-featured operating systems.
- A mailbox is a mutually agreed upon memory location that one or more tasks can use to pass data.
- The tasks rely on the kernel to allow them to write to the location via a post operation or to read from it via a pend operation.

```
void pend(int data, S);

void post(int data, S);
```

- The difference between the pend operation and simply polling the mailbox is that the pending task is suspended while waiting for data to appear. Thus, no time is wasted continually checking the mailbox.

Mailboxes

- The datum that is passed can be a flag used to protect a critical resource (called a key).
- When the key is taken from the mailbox, the mailbox is emptied. Thus, although several tasks can pend on the same mailbox, only one task can receive the key.
- Since the key represents access to a critical resource, simultaneous access is precluded.

Queues

- Some operating systems support a type of mailbox that can queue multiple pend requests.
- The queue can be regarded as any array of mailboxes.
- Queue should not be used to pass array data; pointers should be used instead.
- Queues – control access to the “circular buffer”.

Critical Regions

- Multitasking systems are concerned with resource sharing.
- In most cases, these resources can only be used by one task at a time, and use of the resource cannot be interrupted.
- Such resources are said to be serially reusable.

Critical Regions

- While the CPU protects itself against simultaneous use, the code that interacts with the other serially reusable resources cannot.
- Such code is called a critical region.
- If two tasks enter the same critical region simultaneously, a catastrophic error occur.

Semaphores

- The most common methods for protecting critical regions involves a special variable called a semaphore.
- A semaphore S is a memory location that acts as a lock to protect critical regions.
- Two operations: wait P(S), signal V(S)

Semaphores

- The wait operation suspends any program calling until the semaphore S is FALSE, whereas the signal operation sets the semaphore S to FALSE.
- Code that enters a critical region is bracketed by calls to wait and signal. This prevents more than one process from entering the critical region.

```
void P(int S)
{
    while (S == true);
    S = true;
}

void V(int S)
{
    S = false;
}
```

Semaphore is initialized to false.

Process_1	Process_2
.	.
.	.
.	.
P(S)	P(S)
critical region	critical region
V(S)	V(S)
.	.
.	.
.	.

Mailboxes and Semaphores

- Mailboxes can be used to implement semaphores if semaphore primitives are not provided by the operating system.
- In this case, there is the added advantage that the pend instruction suspends the waiting process rather than actually waiting for the semaphore.

```
void P(int S)
{
    int key = 0;
    pend(key, S);
}

void V(int S)
{
    int key = 0;
    post(key, S);
}
```

Counting Semaphores

- The P and V semaphores are called binary semaphores because they can take one of two values.

- Alternatively, a counting semaphore can be used to protect pools of resources, or to keep track of the number of free resources.

```
void P(int S)
{
    S--;
    while(S < 0);
}

void V(int S)
{
    S++;
}
```

```
void MP(int R)                  /* multiple wait */
{
    P(S);                       /* lock counter */
    R--;                        /* request a resource */
    if (R < 0)                  /* none available? */
    {
        V(S);                   /* release counter */
        P(T);                   /* wait for free resource */
    }
    V(S);                       /* release counter */
}

void MV(int R)                  /* multiple signal */
{
    P(S);                       /* lock counter */
    R++;                        /* free resource */
    if (R <= 0)
        V(T);
    else
        V(S);                   /* release counter */
}
```

Counting Semaphores

- The integer R keeps track of the number of free resources. Binary semaphore S protects R, and binary semaphore T is used to protect the pool of resources.

- The initial value of S is set to False, T to True, and R to the number of available resources in the kernel.

Other Synchronization Mechanisms

- Monitors are abstract data types that encapsulate the implementation details of the serial reusable resource and provides a public interface.
- Instances of the monitor type can only be executed by one process at a time.
- Monitors can be used to implement any critical region.

Other Synchronization Mechanisms

- Event-flag structures allow for the specification of an event that causes the setting of some flag.
- A second process is designed to react to this flag.
- Event flags in essence represent simulated interrupts created by the programmer.

Deadlock

- When tasks are competing for the same set of two or more serially reusable resources, a deadlock situation or deadly embrace may occur.

- Starvation differs from deadlock in that at least one process is satisfying its requirements but one or more are not.

- In deadlock, two or more processes cannot advance due to mutual exclusion.

Deadlock

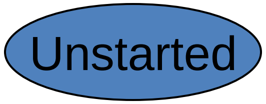
- When tasks are competing for the same set of two or more serially reusable resources, a deadlock situation or deadly embrace may occur.

- Starvation differs from deadlock in that at least one process is satisfying its requirements but one or more are not.

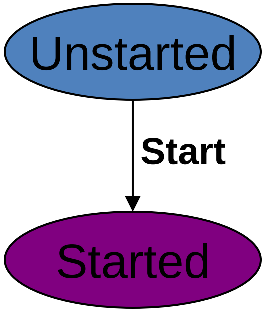
- In deadlock, two or more processes cannot advance due to mutual exclusion.

Serious problem!!!

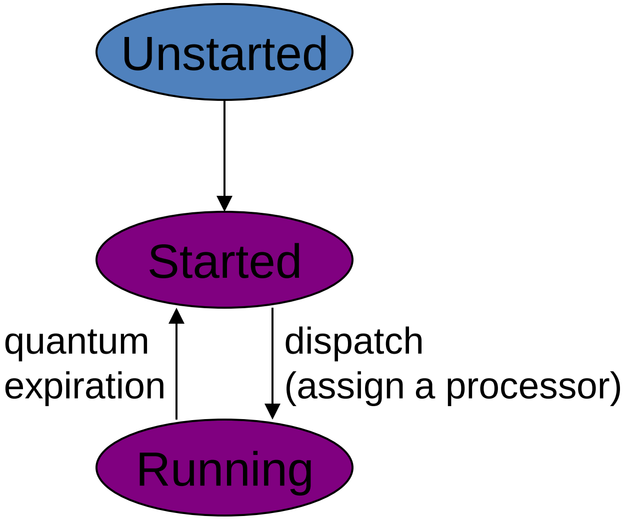
Life Cycle of a Thread



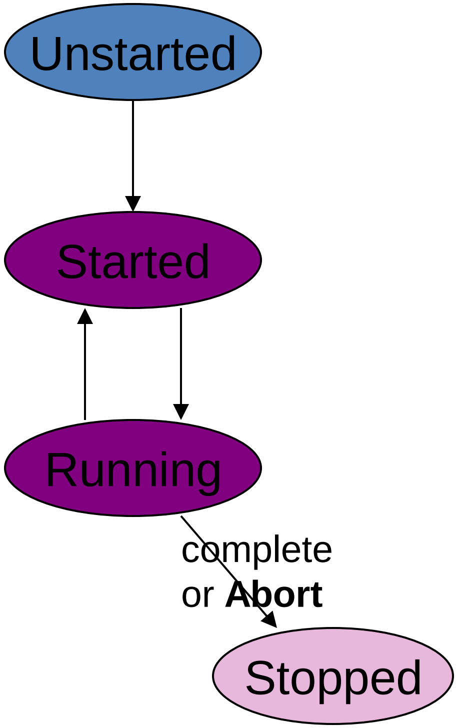
Life Cycle of a Thread



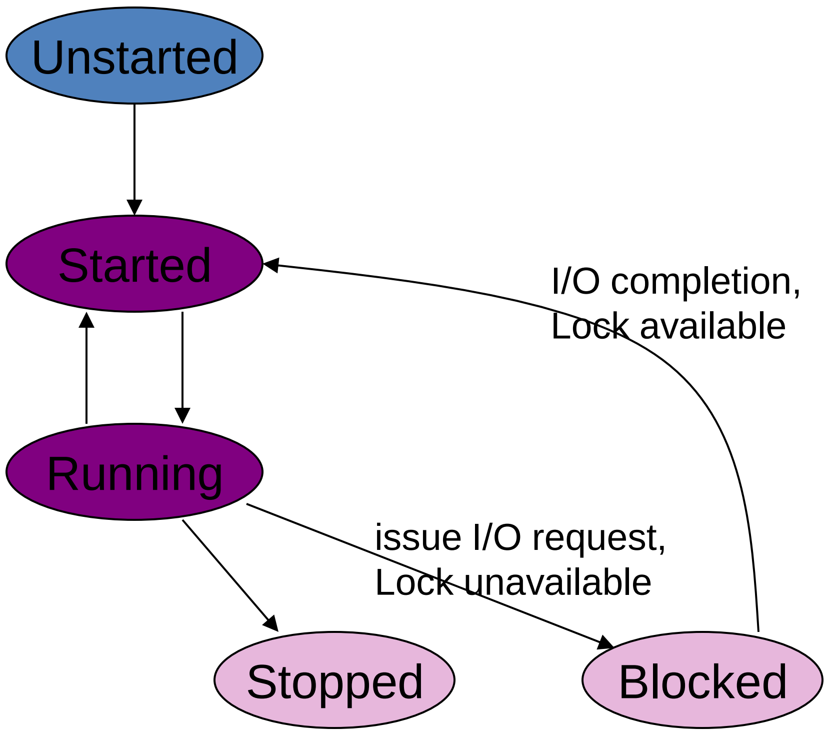
Life Cycle of a Thread



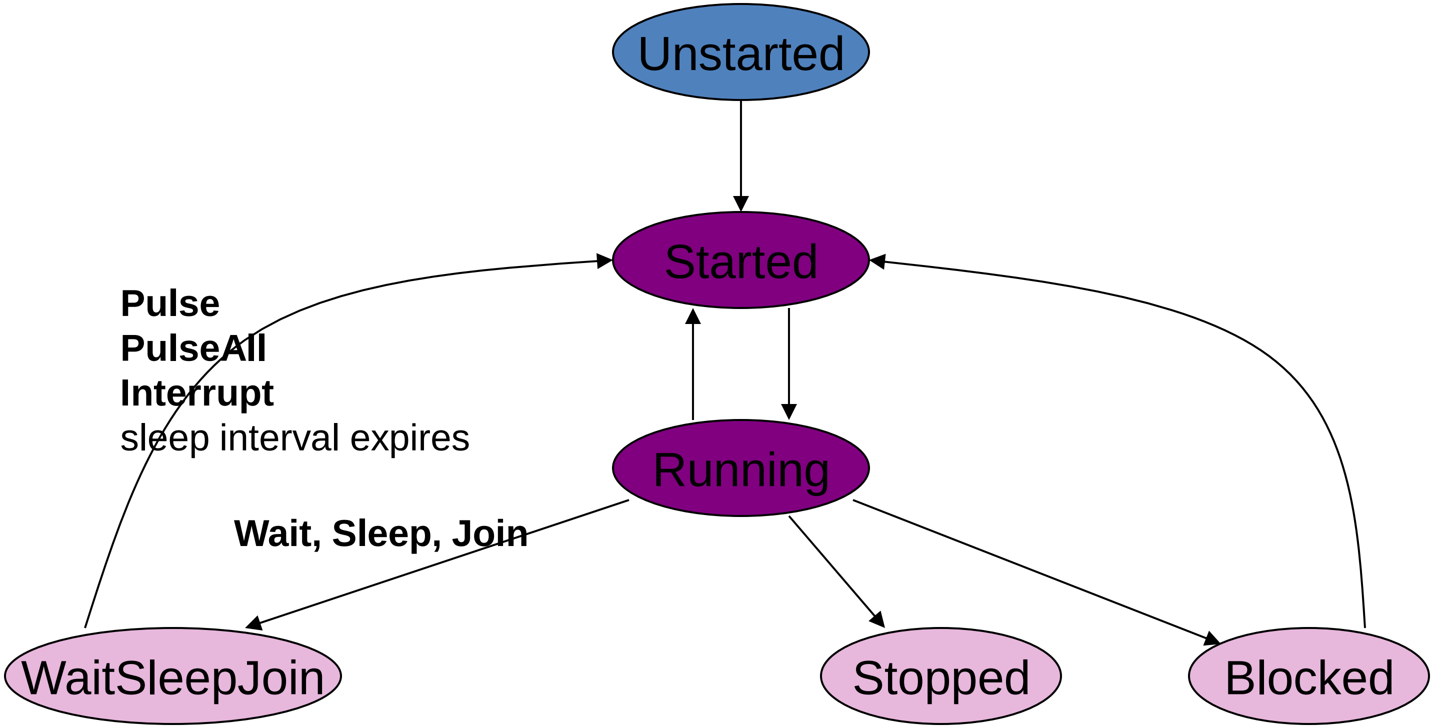
Life Cycle of a Thread



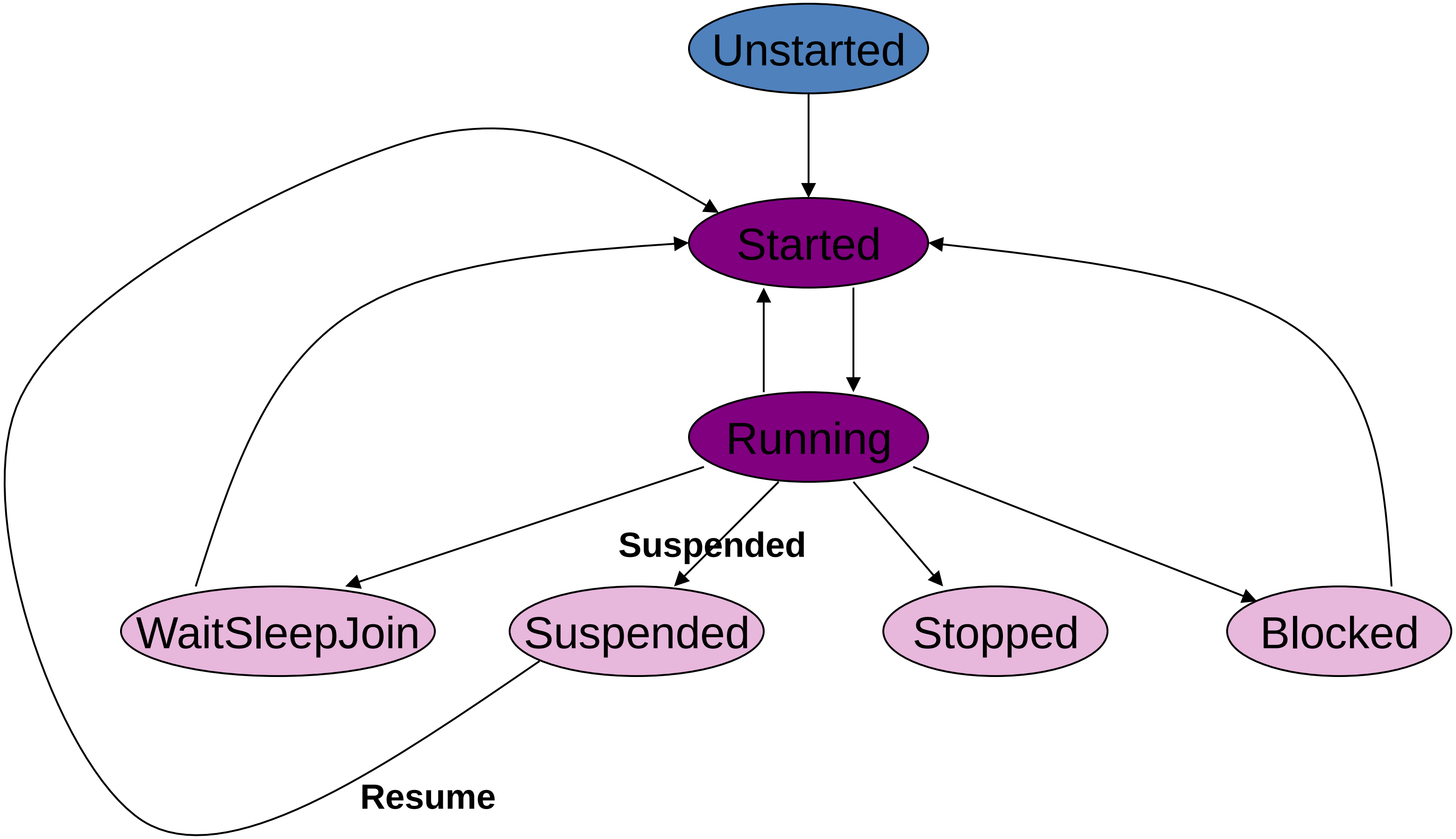
Life Cycle of a Thread



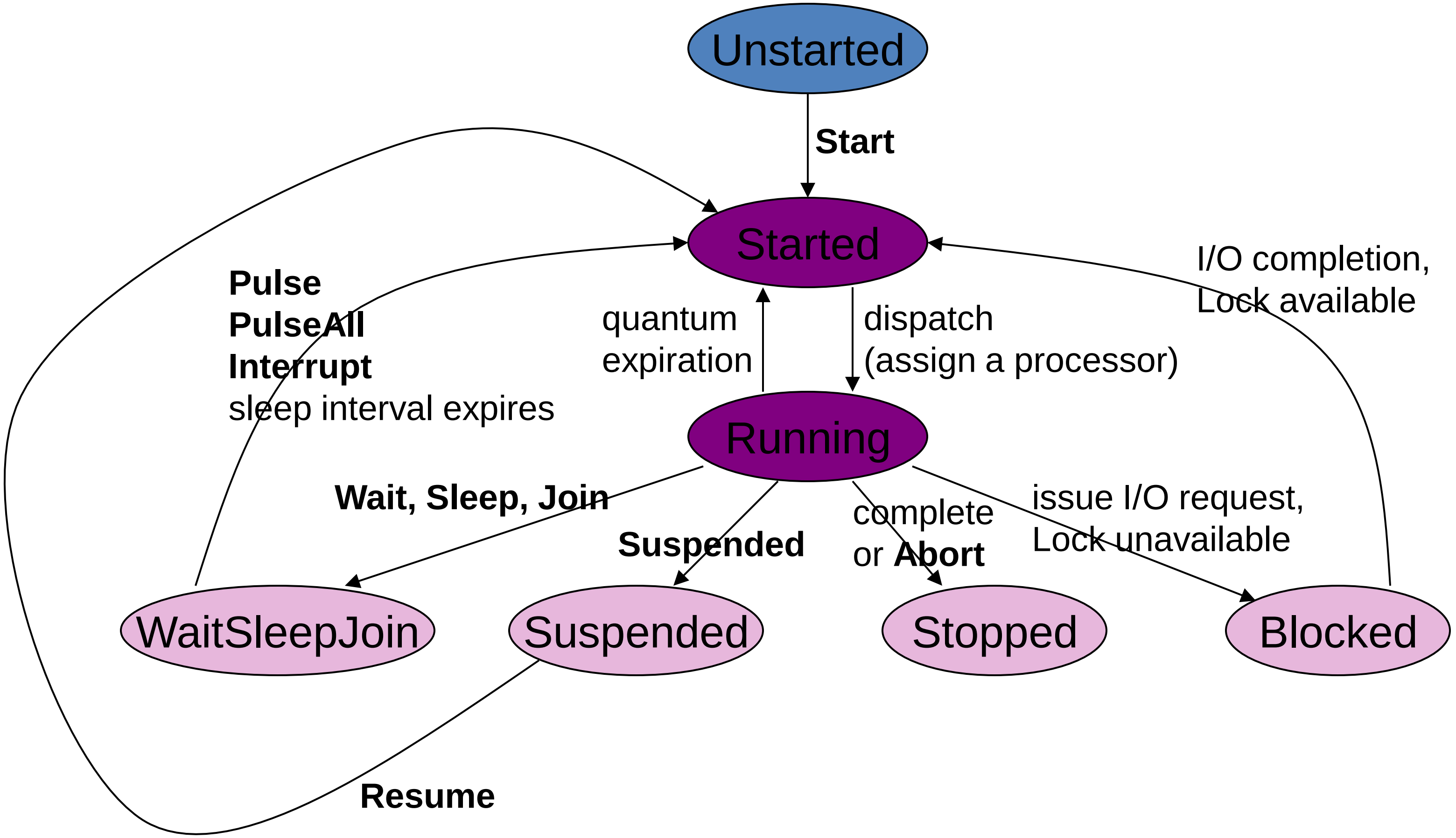
Life Cycle of a Thread



Life Cycle of a Thread



Life Cycle of a Thread



Deadlock Prevention

- Mutual exclusion can be removed through the use of programs that allow resources to appear to be shareable by application (e.g. spoolers for printers).
- To prevent “hold and wait”, we allocate to a process all potentially required resources at the same time.
- Finally, preemption can preclude deadlock. Again, this will create starvation.

Deadlock Avoidance

- The best way to deal with deadlock is to avoid it altogether.
- A lock refers to any semaphore used to protect a critical region.
- For example, if the semaphores protecting critical resources are implemented by mailboxes with time-outs, deadlocking cannot occur, (but starvation of one or more tasks is possible).

Deadlock Avoidance

1. Minimize the number of critical regions as well as minimizing their size.
2. All processes must release any lock before returning to the calling function.
3. Do not suspend any task while it controls a critical region.
4. All critical regions must be error free.
5. Do not lock devices in interrupt handlers.
6. Always perform validity checks on pointers used within critical regions. (Pointer errors are common in C and can lead to serious problems within the critical regions.)

Deadlock Avoidance: The Banker's Algorithm

- Analogy of a bank: depositors and cash reserve.
- The algorithm ensures that the number of resources attached to all processes and potentially needed for at least one to complete, can never exceed the number of resources for the system.
- The program shall not enter "unsafe state" to avoid deadlock.

Generalized Banker's Algorithm

- Extended to two or more pools of resources.
- Consider a set of processes $p_1, \ldots, p_n$ and a set of resources $r_1, \ldots, r_m$.
- max[i,j] represents the max claim of resources type j by process i.
- alloc[i,j] represents the number of units of resources j held by process i.

Generalized Banker’s Algorithm

- c_j : resources of type j
- avail[j] : the resulting number of available resources of type j if the resource is granted.

Generalized Banker's Algorithm

- c_j : resources of type j
- avail[j] : the resulting number of available resources of type j if the resource is granted.

$$avail[j] = c_j - \sum_{0 \leq i < n} alloc[i, j]$$

Generalized Banker's Algorithm

- c_j : resources of tpye j
- avail[j] : the resulting number of available resources of tpye j if the resource is granted.

$$avail[j] = c_j - \sum_{0 \le i < n} alloc[i, j]$$

$$p_i : max[i, j] - alloc[i, j] \le avail[j]$$

$$for \quad 0 \le j < m, 0 \le i < n$$

Generalized Banker's Algorithm

- c_j : resources of tpye j
- avail[j] : the resulting number of available resources of tpye j if the resource is granted.

$$avail[j] = c_j - \sum_{0 \le i < n} alloc[i, j]$$

$$p_i : max[i, j] - alloc[i, j] \le avail[j]$$
$$for \quad 0 \le j < m, 0 \le i < n$$

If no such p_i exists, the state is unsafe.

Priority Inversion

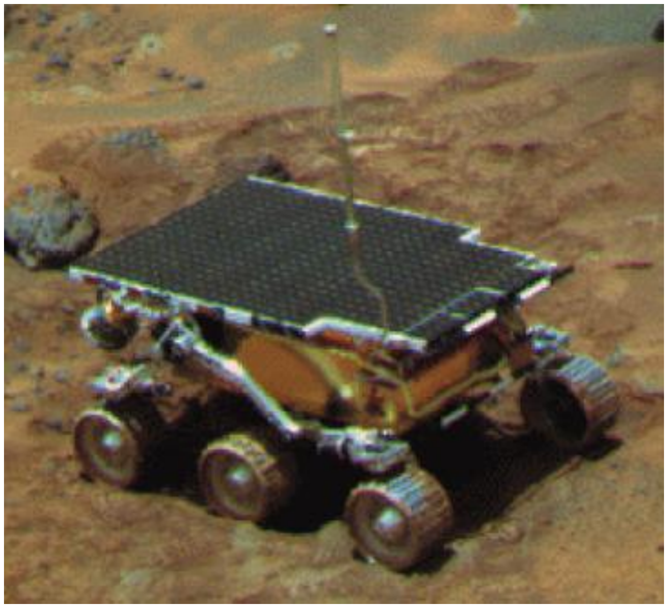
- When a low-priority task blocks a higher-priority one, a priority inversion is said to occur.

- The problem of priority inversion in real-time systems has been studied intensively for both fixed-priority and dynamic-priority scheduling.

Priority Inheritance Protocol

- The priority of tasks are dynamically changed so that the priority of any task in a critical region gets the priority of the highest task pending on that same critical region.
- When a task blocks one or more higher-priority tasks, it temporarily inherits the highest priority of the blocked tasks.

Priority Inheritance Protocol



- A 1997 NASA incident of Mars Pathfinder Space mission's *Sojourner* rover vehicle: A meteorological data-gathering task (low priority low frequency) blocked a communications task (high priority high frequency). This infrequent scenario caused the system to reset.

- The problem was diagnosed in ground-based testing and remotely corrected by reenabling the priority inheritance mechanism.

Priority Inheritance Protocol

- Priority Inheritance Protocol does not prevent deadlock. In fact, PIP can cause deadlock or multiple blocking.
- Priority Ceiling Protocol, which imposes a total ordering on the semaphore access, can get around these problems.

Priority Ceiling Protocol

- Each resource is assigned a priority (the priority ceiling) equal to the priority of the highest priority task can use it.
- A task, T, can be blocked from entering a critical section if there exists any semaphore currently held by some other task whose priority ceiling is greater than or equal to the priority of T.

Memory Management

- Dynamic memory allocation is important in both the use of on-demand memory by applications and the requirements of the operating system.
- Application tasks use memory explicitly through requests for heap memory, and implicitly through the maintenance of the run-time memory needed to support sophisticated high-order languages.
- Operating system needs to perform extensive memory management.

Process Stack Management

- In a multitasking system, context for each task needs to be saved and restored in order to switch processes.

- Run-time stacks work best for interrupt-only systems and foreground/background systems.

- Task-control block model works best with full-featured real-time operating systems.

Run-Time Stack

- A run-time stack is to be used to handle the run-time saving and restoring of context.
- The save routine is called by an interrupt handler to save the current context of the machine into a stack area.
- To prevent disaster, save call should be made immediately after interrupts have been disabled.

Run-Time Stack

- The restore routine is called by an interrupt handler to restore the context of the main machine from a stack area.

- The restore routine should be called just before interrupts are enabled and before returning from the interrupt handler.

Run-Time Stack

save (stack)

```
DPI
STORE    R0, &stack, I
LOAD     R0, &stack
ADD      R0, 1
STORE    R1, R0, I
ADD      R0, 1
STORE    R2, R0, I
ADD      R0, 1
STORE    R3, R0, I
ADD      R0, 1
STORE    PC, R0, I
ADD      R0, 1
STORE    R0, &stack
EPI

RETURE
```

restore (stack)

```
DPI
LOAD     R0, &stack
SUB      R0, 1
LOAD     PC, R0, I
SUB      R0, 1
LOAD     R3, R0, 1
SUB      R0, 1
LOAD     R2, R0, I
SUB      R0, 1
LOAD     R1, R0, I
STORE    R0, &stack
SUB      R0, 1
LOAD     R0, R0, I
EPI

RETURE
```

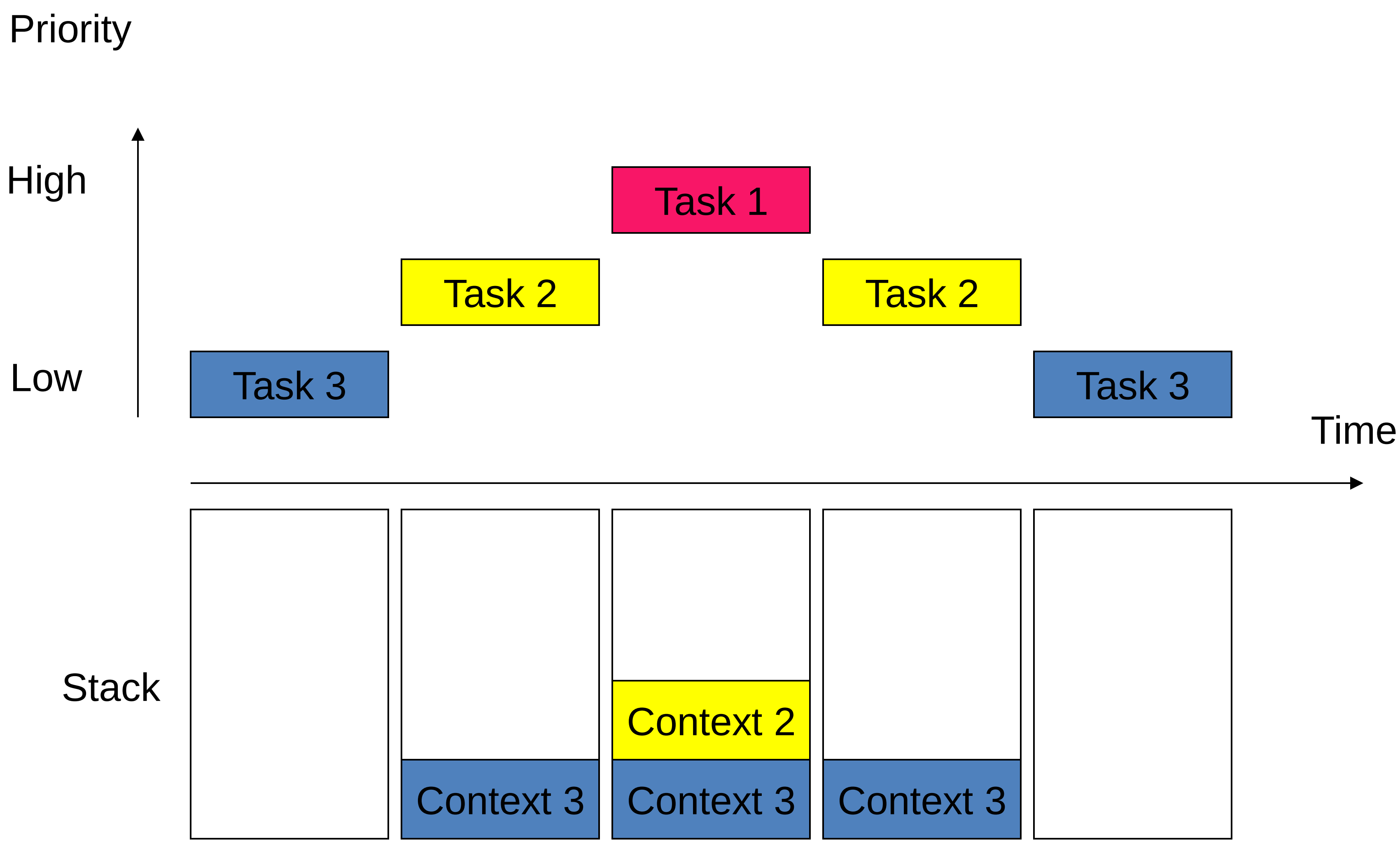
Run-Time Stack

```
void int_handler (void)
{
    save(mainstack);
    switch(interrupt)
    {
        case 1: int1();
                break;
        case 2: int2();
                break;
    }
    restore(mainstack);
}
```

```
void int1(void)
{
    save(stack);
    task1();
    restore(stack);
}

void int2(void)
{
    save(stack);
    task2();
    restore(stack);
}
```

Run-Time Stack



Task-Control Block Model: Fixed Case

- N task-control blocks are allocated at system generation time, all in the dormant state.
- As tasks are created, the task-control block enters the ready state.
- Prioritization or time slicing will move the task to the execute state.
- If a task is to be deleted, its task-control block is simply placed in the dormant state.

Task-Control Block Model: Dynamic Case

- In the dynamic case, task-control blocks are added to a linked list as tasks are created.
- The tasks are in the suspended state upon creation and enter the ready state via an operating system call.
- The tasks enter the execute state owing to priority or time slicing.
- When a task is deleted, its task-control block is removed from the linked list, and its heap memory allocation is returned to the unoccupied status.

Run-Time Ring Buffer

- A run-time stack cannot be used in a round-robin system because of its FIFO nature of scheduling.
- A circular queue can be used in a round-robin system to save context.
- The context is saved to the tail of the list and restored from the head of the list.

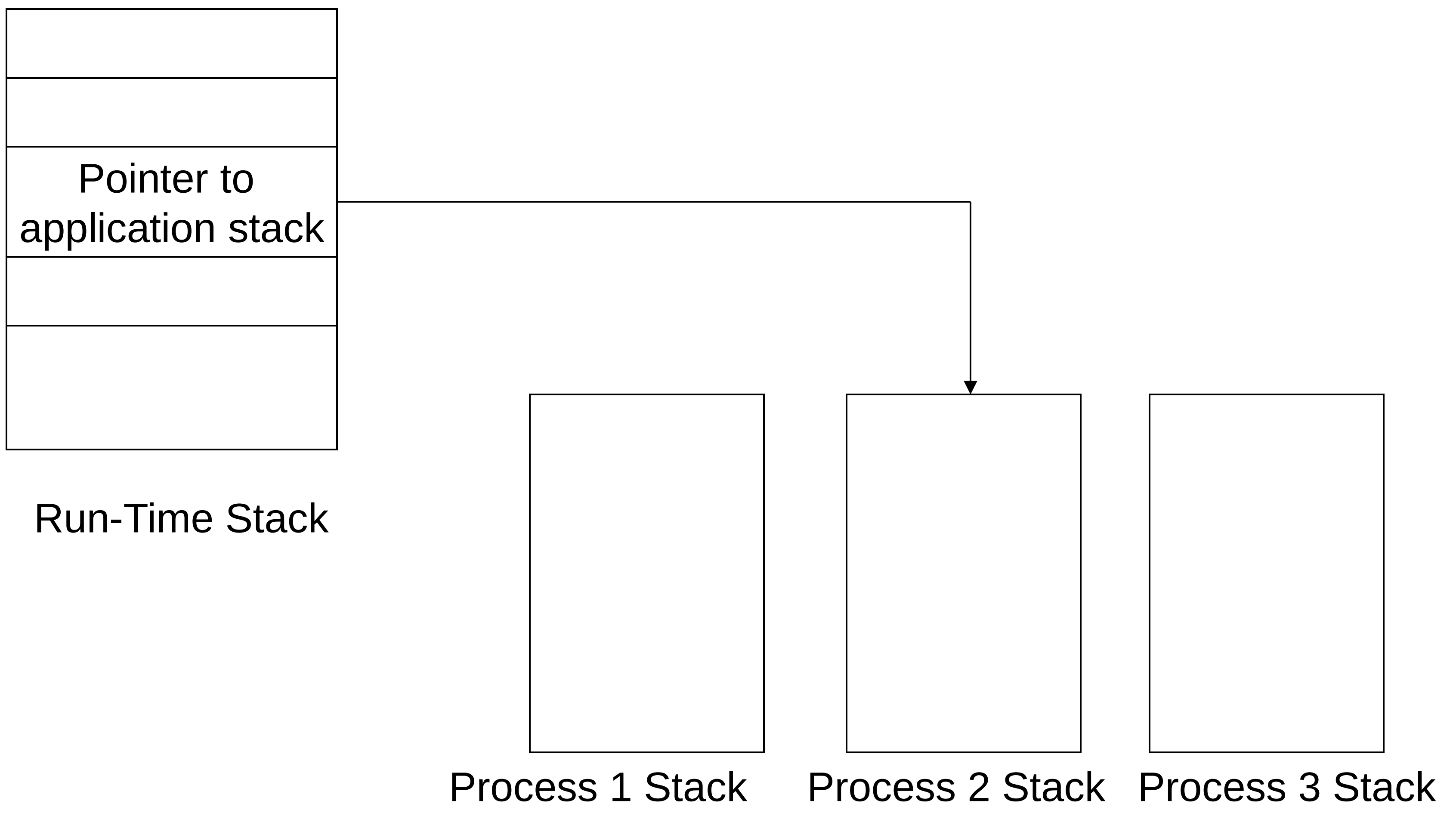
Maximum Stack Size

- The maximum amount of space needed for the run-time stack needs to be known *a priori*.
- In general, stack size can be determined if recursion is not used and heap data structures are avoided.
- Ideally, provision for at least one more task than anticipated should be allocated to the stack to allow for spurious interrupts.

Multiple-Stack Arrangement

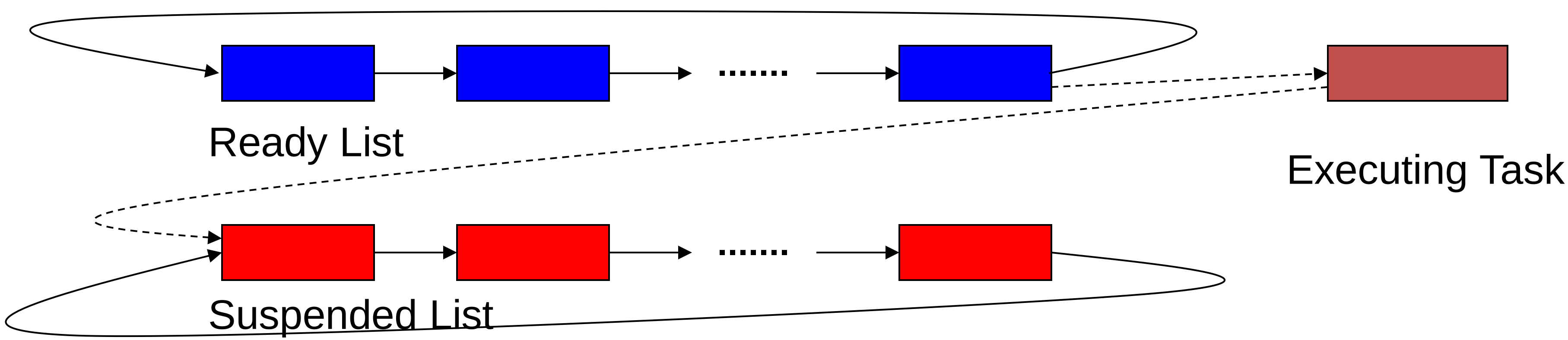
- Often a single run-time stack is inadequate to manage several processes, e.g. in a foreground/background system.
- A multiple-stack scheme uses a single run-time stack and several application stacks.
- The embedded real time system using multiple stacks can be implemented by a language that supports reentrancy and recursion, such as C.

Multiple-Stack Arrangement



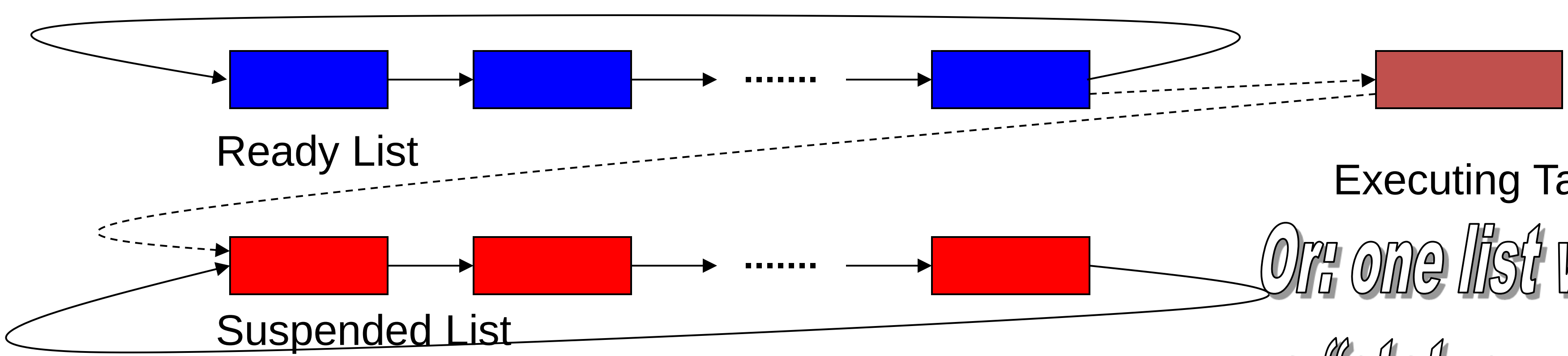
Memory Management in the Task-Control-Block Model

- When implementing the TCB model of real-time multitasking, the chief memory management issue is the maintenance of the linked lists for the ready and suspended tasks.



Memory Management in the Task-Control-Block Model

- When implementing the TCB model of real-time multitasking, the chief memory management issue is the maintenance of the linked lists for the ready and suspended tasks.

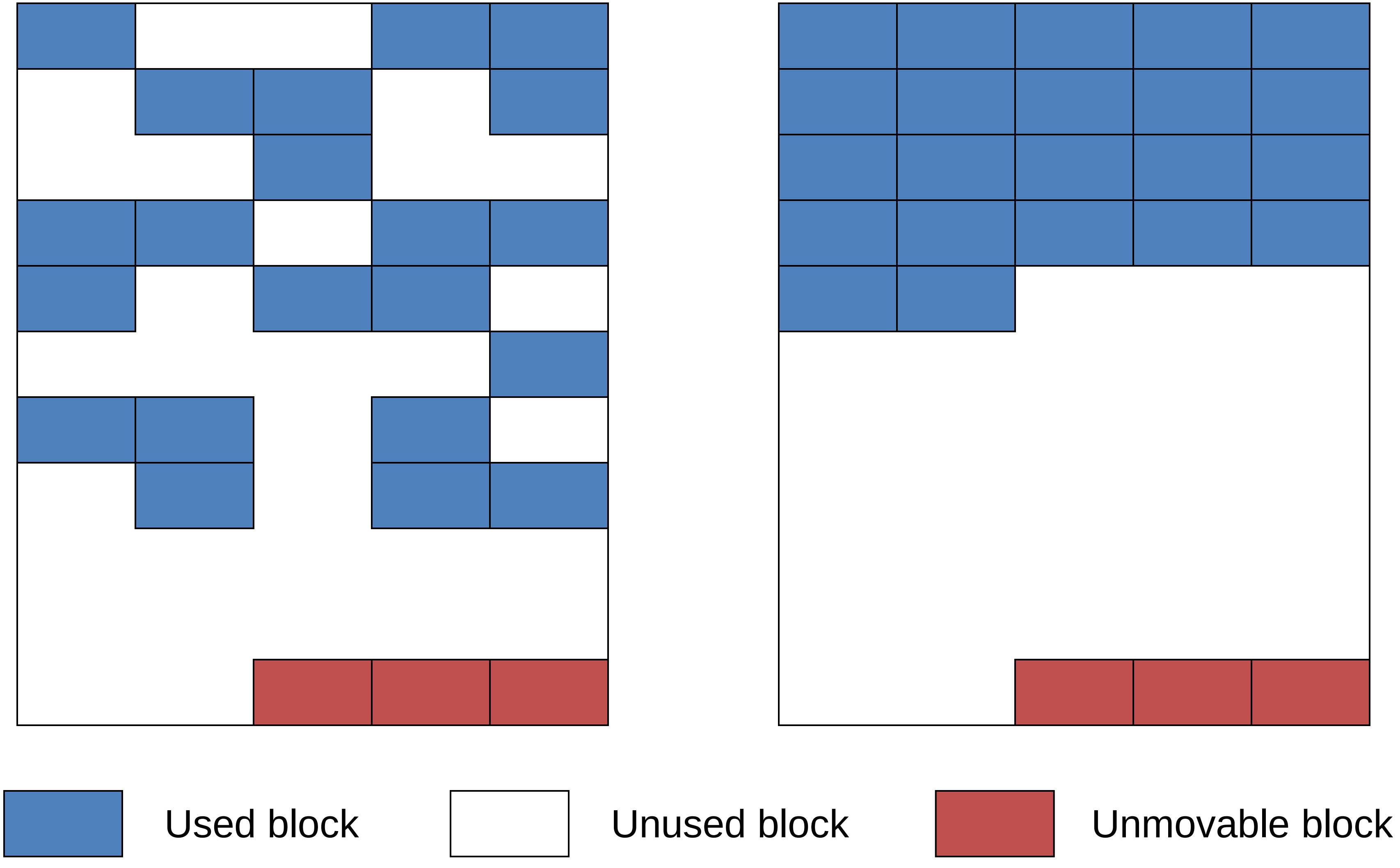


Or: one list with a "status word"

Dynamic Memory Allocation

- Usually, memory fragmentation problem can be solved by so-called "garbage collection" (defragmentation) software.

Dynamic Memory Allocation



Dynamic Memory Allocation

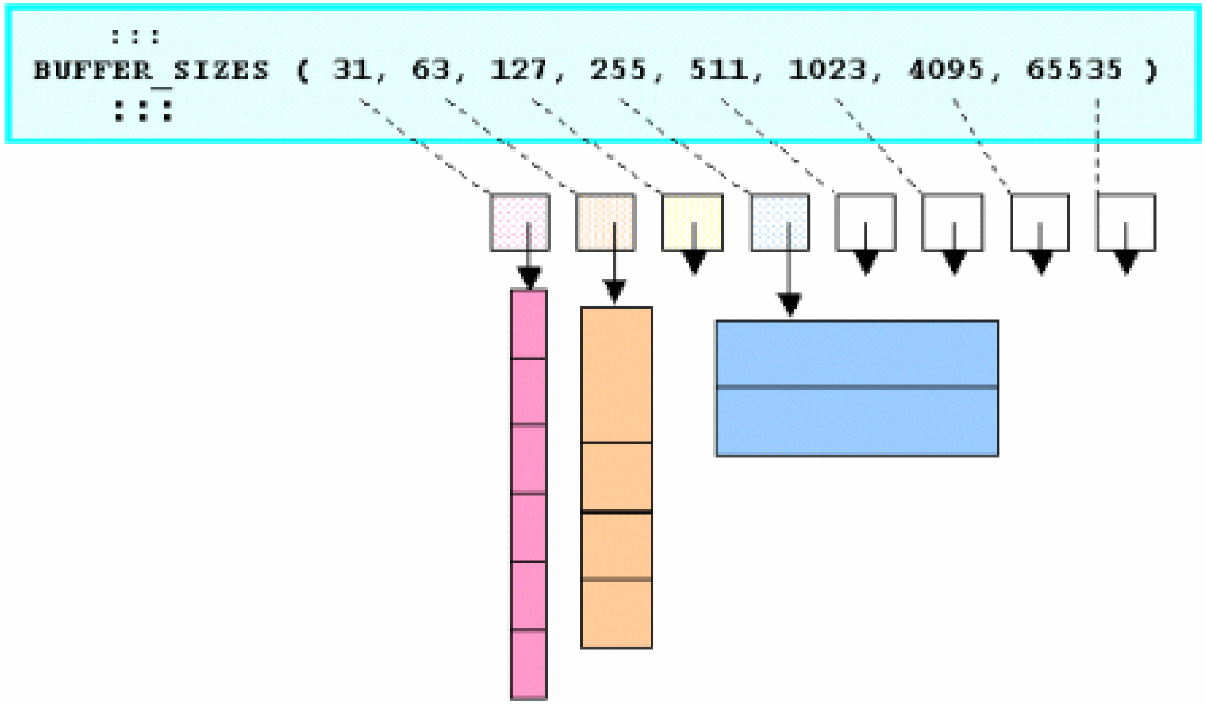
- Usually, memory fragmentation problem can be solved by so-called "garbage collection" (defragmentation) software.

- Unfortunately, "garbage collection" algorithms are often wildly non-deterministic – injecting randomly-appearing random-duration delays into heap services.

Dynamic Memory Allocation

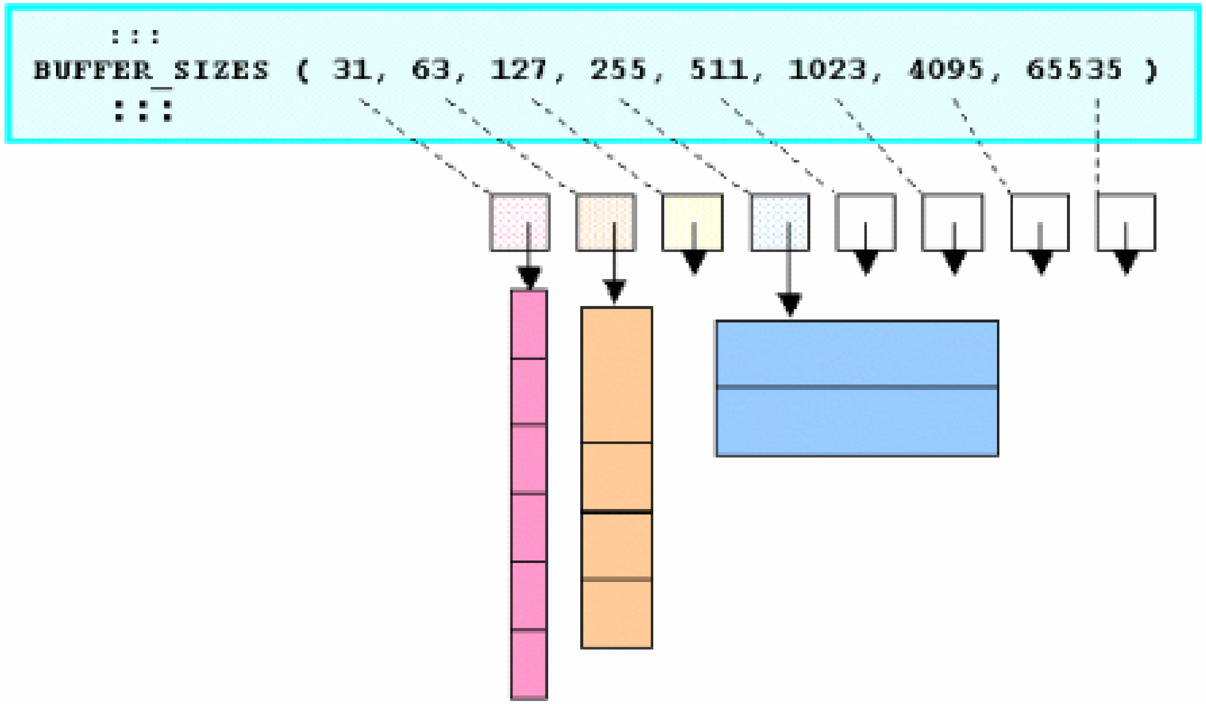
- RTOSs offer non-fragmenting memory allocation techniques instead of heaps.
- They do this by limiting the variety of memory chunk sizes they make available to application software.
- While this approach is less flexible than the approach taken by memory heaps, they do avoid external memory fragmentation and avoid the need for defragmentation.

Dynamic Memory Allocation



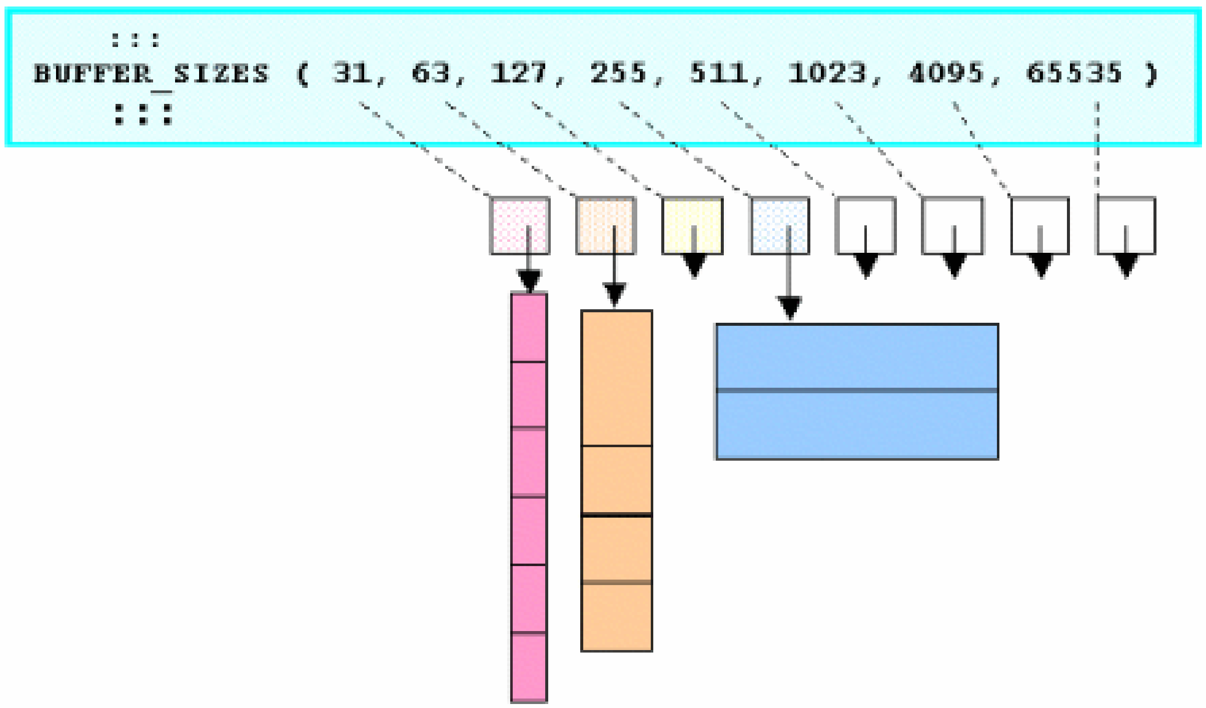
- Pools totally avoid external memory fragmentation, by not permitting a buffer that is returned to the pool to be broken into smaller buffers in the future.

Dynamic Memory Allocation



- Instead, when a buffer is returned the pool, it is put onto a "free buffer list" of buffers of its own size that are available for future re-use at their original buffer size.

Dynamic Memory Allocation



- Memory is allocated and de-allocated from a pool with deterministic, often constant, timing.