Chapter 3

PROCESSES AND THREADS

3.1 EVOLUTION OF OPERATING SYSTEMS 3-2

3.2 PROCESSES 3-5
3.2.2 Structure of a Process 3-5
3.2.3 Process States 3-6
3.2.4 Process Control Block (PCB) 3-9
3.2.5 Process Queues 3-9
3.2.6 Prioritized Ready Queue 3-11

3.3 PROCESS SWITCHING 3-12
3.3.1 Process Preemption 3-13
3.3.2 Dispatcher Primitives 3-14
3.3.3 Time Quantum 3-16
3.3.4 Blocking Event 3-17
3.3.5 I/O Interrupt or Deblocking Event 3-18

3.4 THREADS 3-18

3.4.1 Structure of Threads 3-19
3.4.2 Thread Example 3-21
3.4.3 Manual Page for CreateThread() System Function 3-24
3.4.4 Example of Usage of Threads 3-27

3.5 THREAD SCHEDULING 3-28
3.5.1 CPU and I/O Bursts 3-29
3.5.3 Scheduler Criteria 3-30
3.5.4 First-Come, First-Served Scheduling 3-31
3.5.5 Shortest-Job-First Scheduling 3-32
3.5.6 Priority Scheduling 3-34
3.5.7 Round-Robin Scheduling 3-35
3.5.8 Multiprocessor Scheduling 3-37
3.5.2 Scheduler Hierarchy 3-38

Copyright © 1998-2002 by Marko Vuskovic

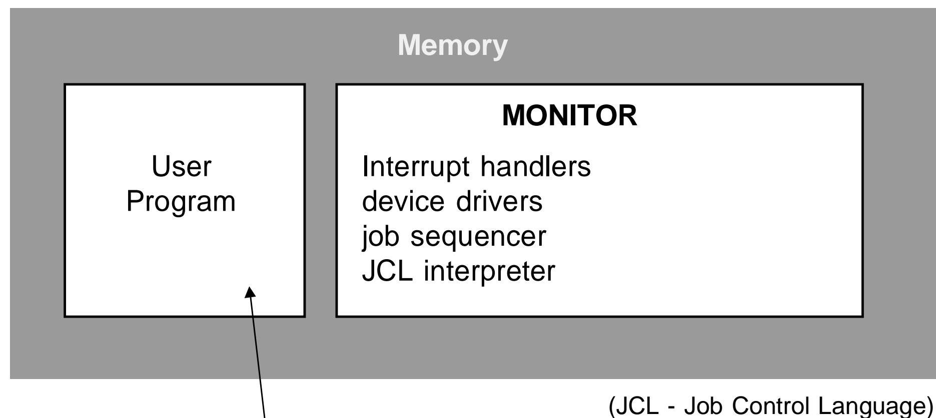
EVOLUTION OF OPERATING SYSTEMS

Serial Processing

Jobs scheduled manually (time slice: k*0.5 hours)
Each job required mounting/dismounting tapes or manipulating card decs.

Simple Batch Processing

Mid 1950's (IBM 701, 704,...)
Operating system consisted of a resident monitor and utilities.
Human operator submits the user's jobs.
Monitor does loading and execution of programs.
Monitor was essentially the only program run by the system.
CPU is idle most of the time (waiting for slow I/O)



(JCL - Job Control Language)

Only one program in memory

Upgrades:

Memory protection (user can't alter monitor area)
Timer (to prevent monopolizing of resources)
Privileged instructions (user program can't read next job.)

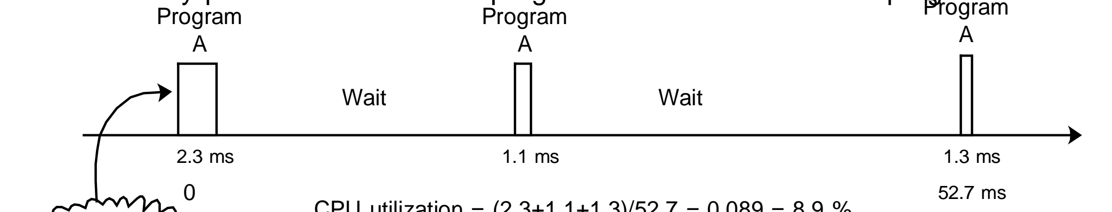
Copyright © 1998-2002 by Marko Vuskovic

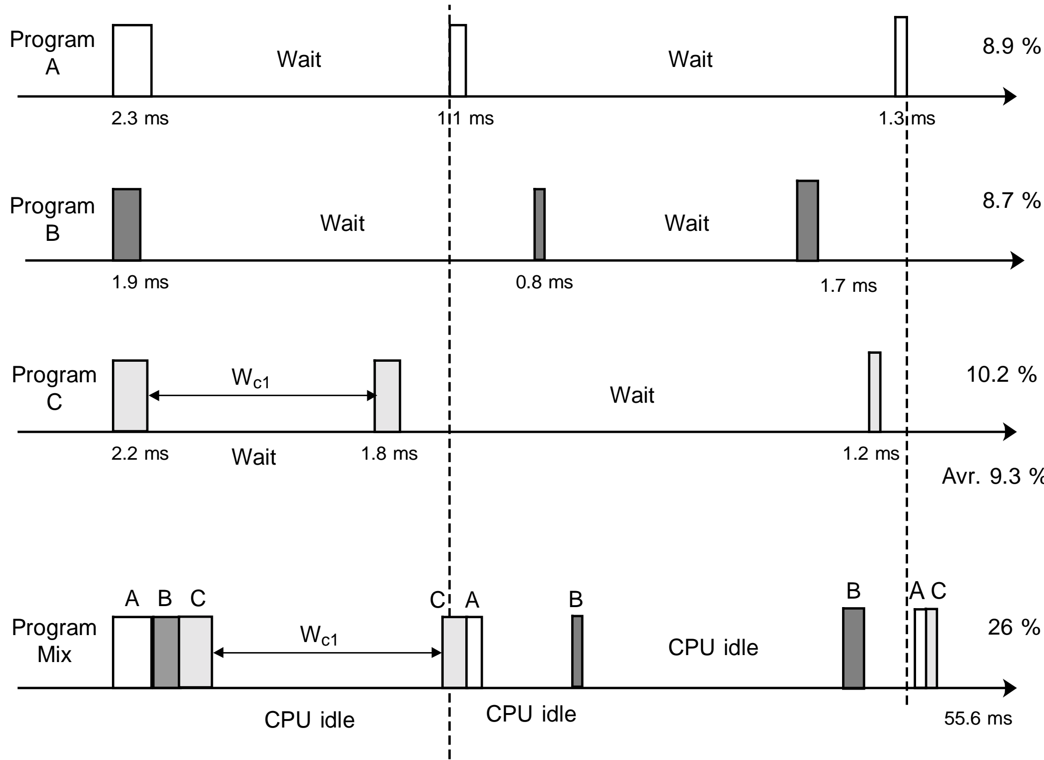
EVOLUTION OF OPERATING SYSTEMS (Cont.)

Multiprogrammed Batch System

(Also called "Multitasking" system)
Several programs are memory resident.
If one program is blocked by I/O, the other program is given the CPU.
Goal: maximize CPU utilization, or maximize number of jobs processed in a time unit.
Basic concepts that enabled this:
I/O interrupt - CPU issues an I/O request and continues to execute another program
I/O processors (channels) and DMA (discussed later in chapter 6) - data transfer with minimal usage of the CPU

Memory protection - Prevent a program to interfere with other programs and the monitor.



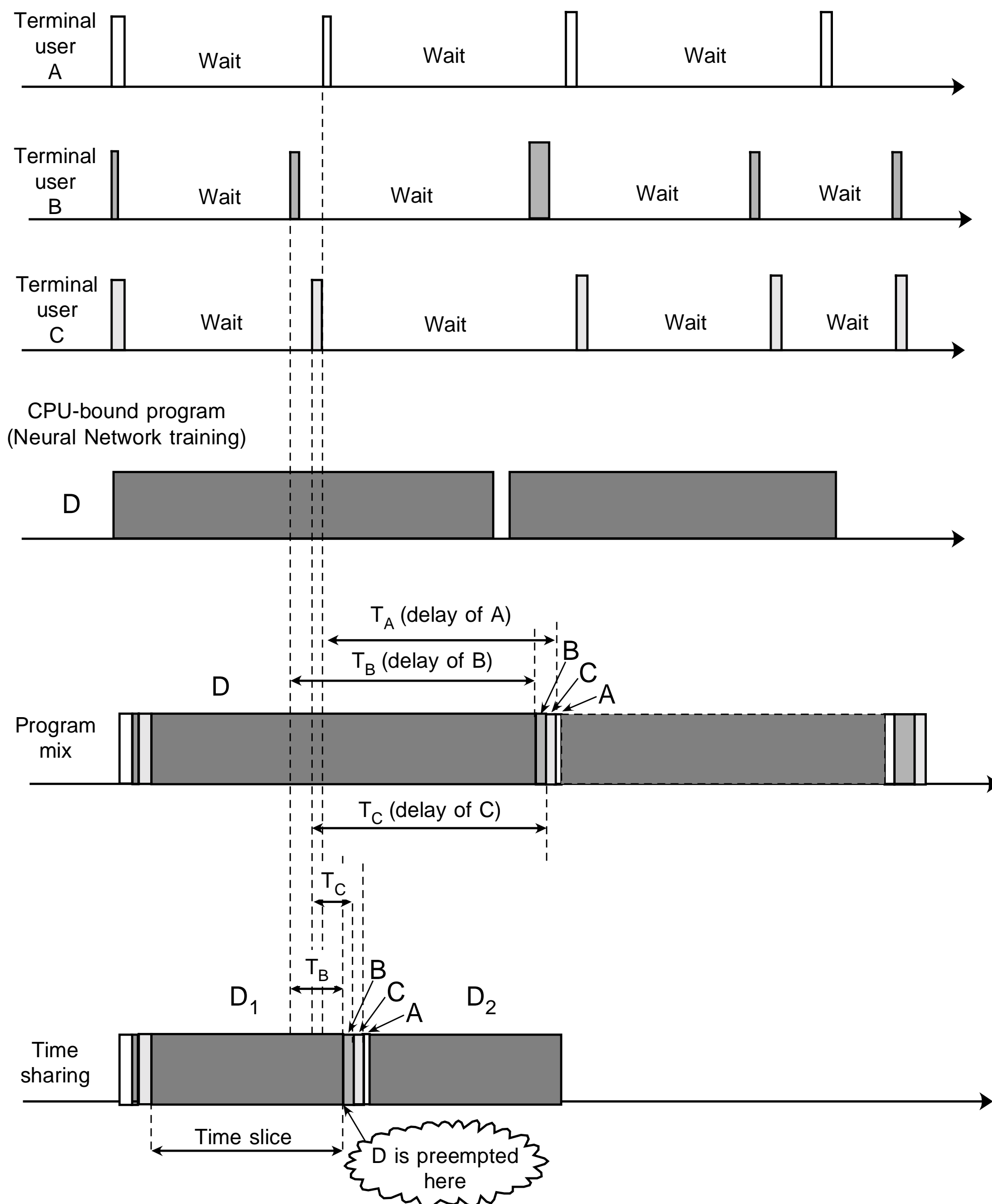


Copyright © 1998-2002 by Marko Vuskovic

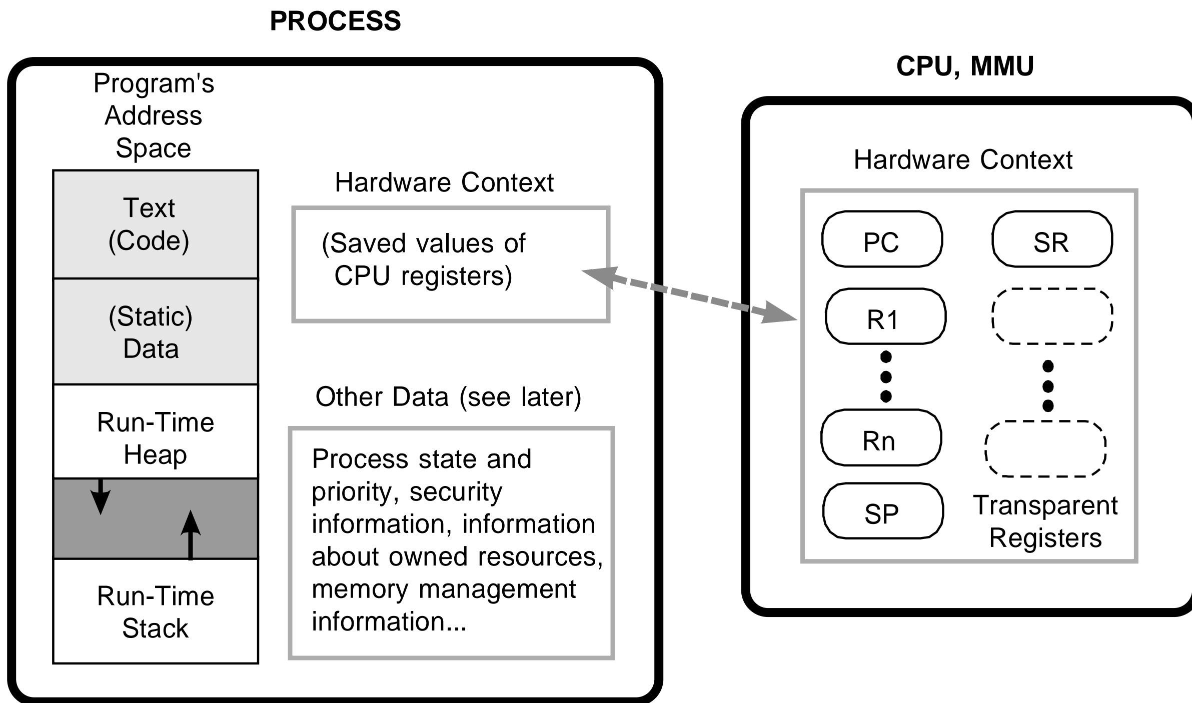
EVOLUTION OF OPERATING SYSTEMS (Cont.)

Time Sharing Systems

For many jobs it is desirable to be interactive (transaction processing, development,...)
Slow human reaction time.
In order to stop CPU-bound jobs from monopolizing the CPU, they have to be preempted.
Goal: Minimize the response time.
First system: Compatible Time-Sharing System (CTSS), MIT, Group MAC.
First commericial: IBM 709, 1961, then IBM 7094.



Copyright © 1998-2002 by Marko Vuskovic



"Process" Is an abstract (intuitive) concept. Here are some deffinitions that can be found in literature:

- Process is a program in execution.
- Process is execution of a program.
- Process is operating system abstraction to represent what is needed to run a single program (Traditional UNIX definition)
- Process is a sequential stream of execution in its own address space.
- Custer: Process is the highest level of abstraction which comprises an executable program (code and data), a private address space, system resources (semaphores, communication ports, files,...)

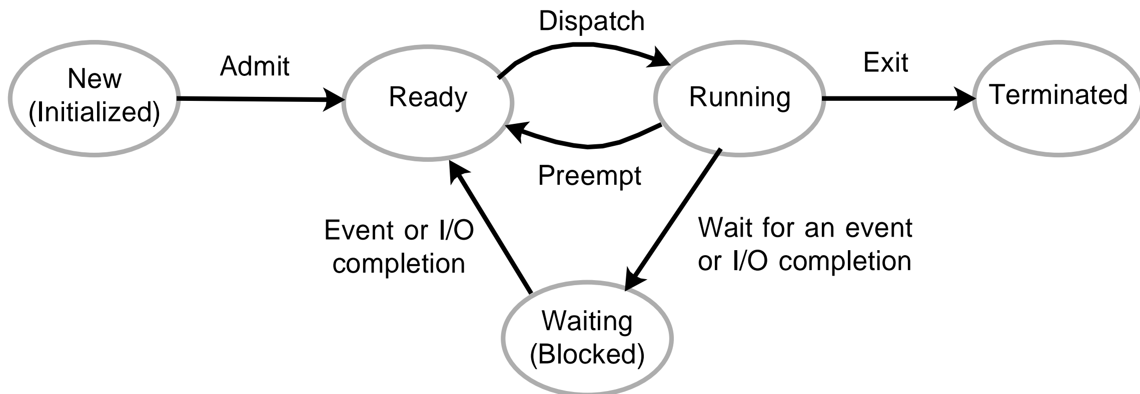
Address space is a set of addresses that the program can reach.

COMMENT: There is often a confusion about the term "Hardware Context". The real hardware context are CPU and MMU registers which contain information relevant to the currently executing process. We can say that a process executes in the context of the CPU and MMU registers, i.e. in its hardware context. However, the term is also used for the data structure in which the CPU and MMU registers are saved when a process is removed from the running state.

Copyright © 1998-2002 by Marko Vuskovic

PROCESS STATES

Five State Model



→ New: New P is created (New batch job, interactive log on, cretaed by the OS to provide a service, spawned by the running P)

New → Rady:
System is prepared to take a new P (number of Ps in the system is limited in order to mantain a performance level.)

Ready → Running:
Processor is freed and available to run a new P.

Running → Terminated:
The running P has exited (completed), aborted or its parent P has terminated. (Terminated Ps are kept in the system typically for accounting purposes.)

Running → Blocked:
P must wait for a resource (file, shared memory), P has initiated an I/O operation, P must synchronize with another P, i.e. wait for that another P completes certain operation.

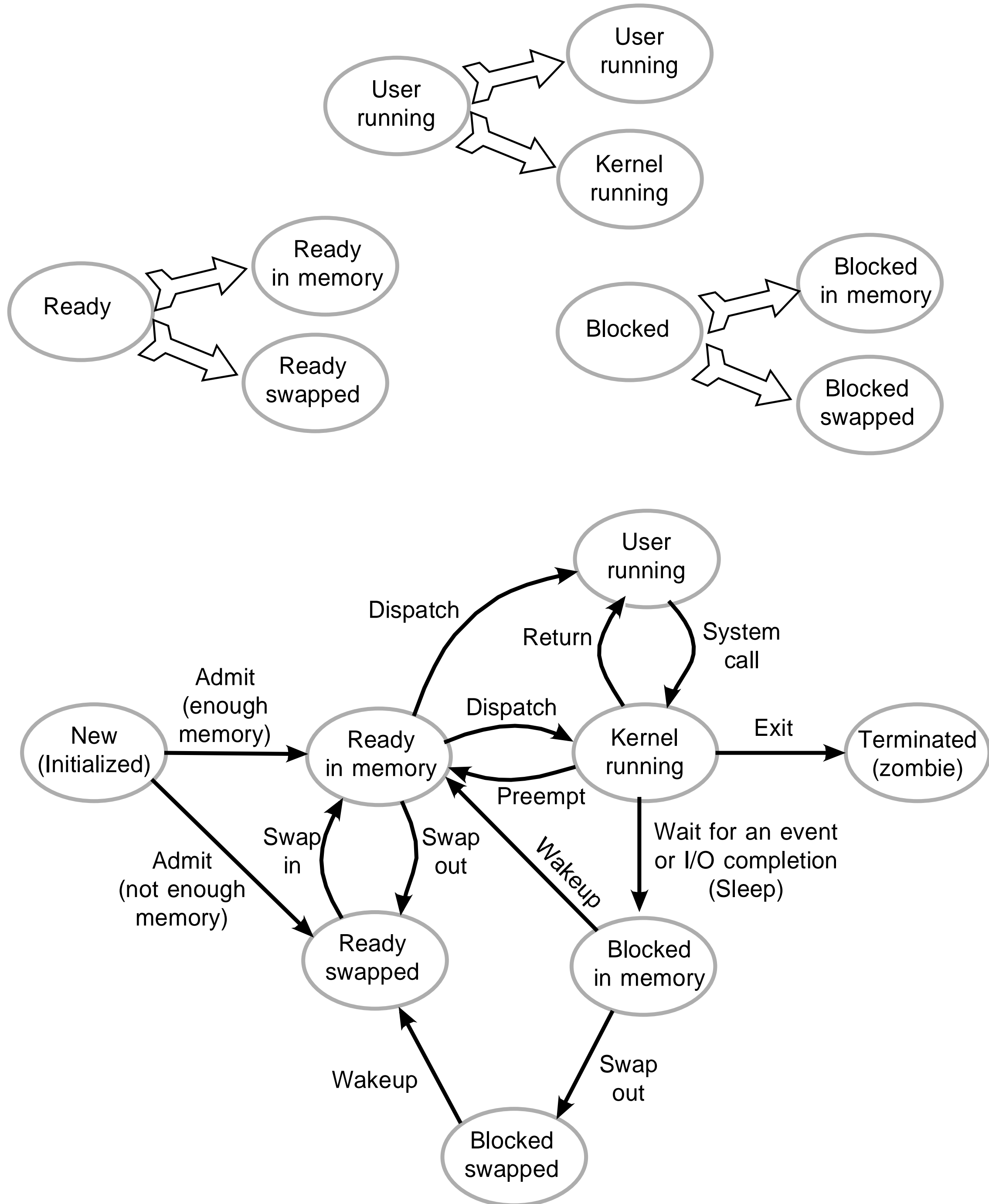
Blocked → Ready:
The event that P has waited has occured.

Running → Ready:
Running P has used its time quantum, or another P with higher priority entered the ready state (in both cases the running P is preempted.)

Copyright © 1998-2002 by Marko Vuskovic

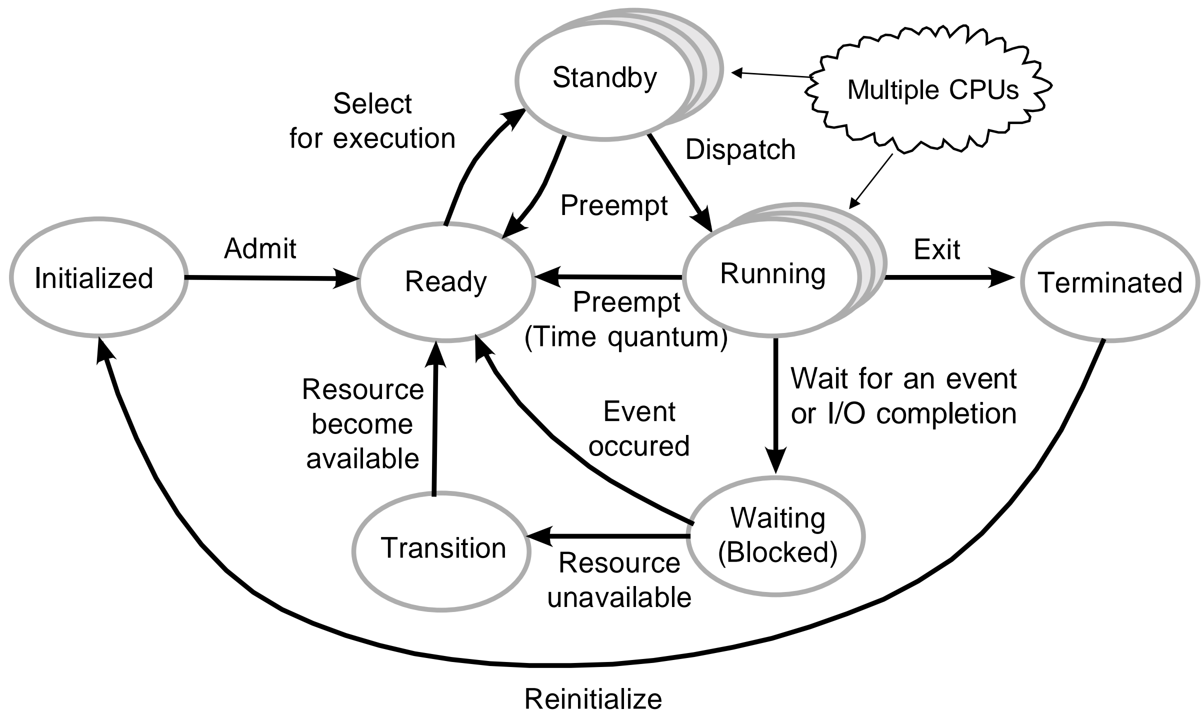
REFINEMENT: UNIX

(Adding new states gives more room for optimization of the process scheduling, i.e. of the system performance)



Copyright © 1998-2002 by Marko Vuskovic

REFINEMENT: WINDOWS NT



NOTICES:

In Windows NT the entities subject to scheduling are threads, not processes (see later)

There are as many standby and running states as the number of CPUs in the system. For each CPU there can be only one thread in standby and one thread in running state.

Thread is in standby state when it is selected to run on a particular processor but the conditions for the context switch are not met yet.

Transition state is typically used for threads whose parts, the program image, a DLL, run-time stack etc. are currently swapped out. (These parts can be swapped out as a whole or partially, see paging and segmentation.)

Copyright © 1998-2002 by Marko Vuskovic

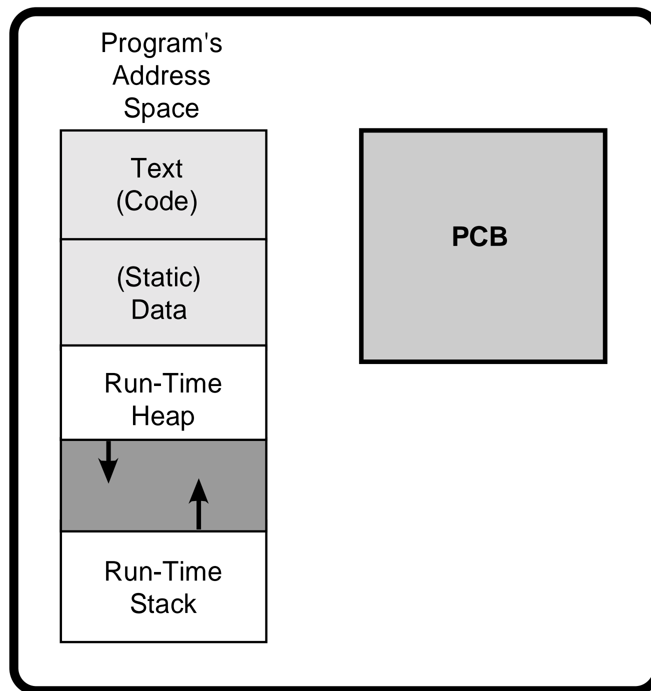
PROCESS CONTROL BLOCK (PCB)

PCB is a data structure which contains all information needed to schedule and run a P.
(In Windows NT this information is divided into two parts: process object and thread object.)

The information in PCB is also called P attributes:

Identifiers:
  P ID, parent's ID, user ID
State attributes:
  P state (running, ready, blocked, terminated...)
  Hardware context (visible and transparent registers, their saved values)
Control information:
  P priority
  pointer to the next PCB in the process queue
  quota limits
Memory management information
Accounting information
I/O information:
  list of devices allocated to the P (open files, drives, synchronization resources)
Security information

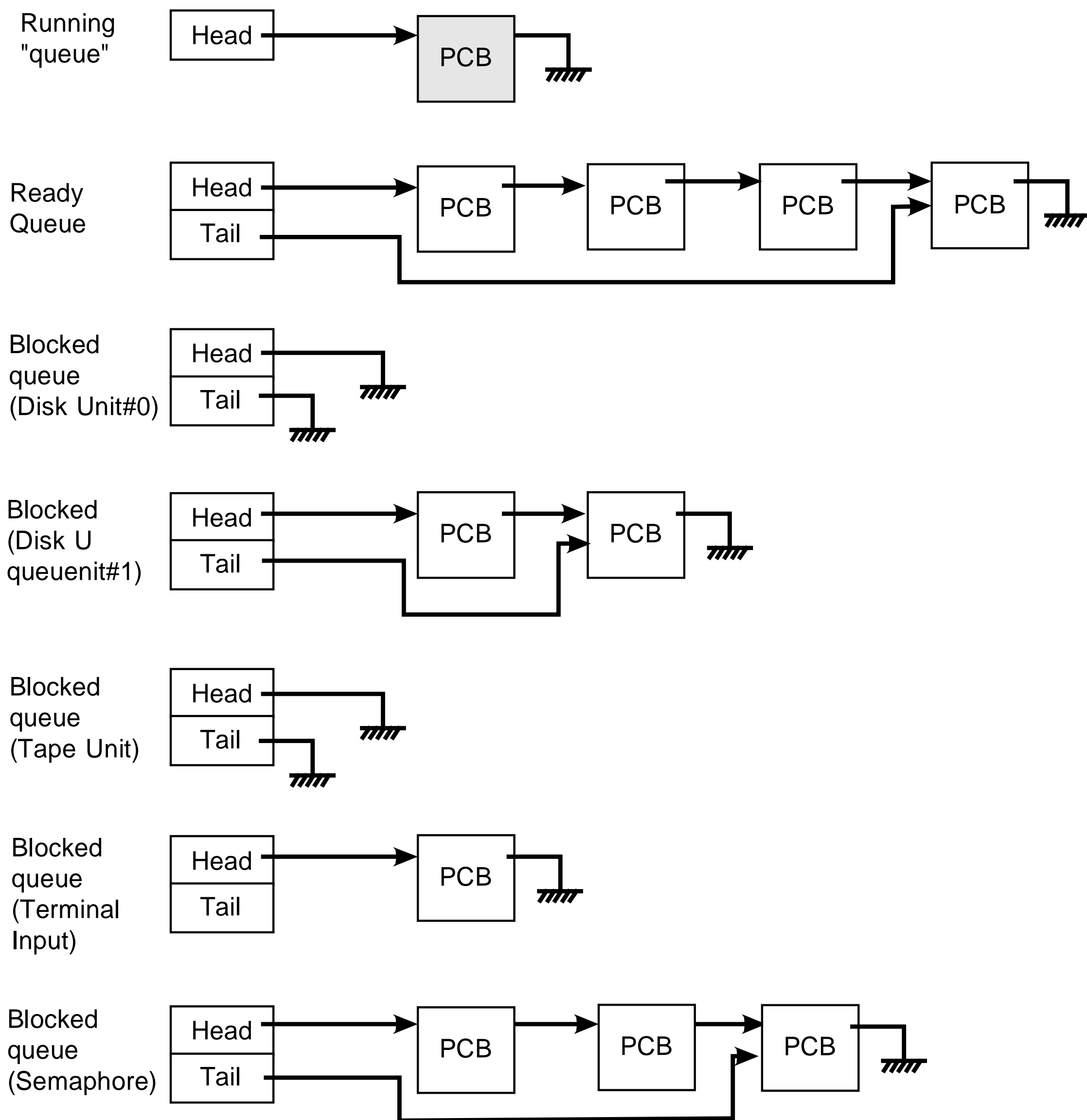
PROCESS



Copyright © 1998-2002 by Marko Vuskovic

PROCESS QUEUES

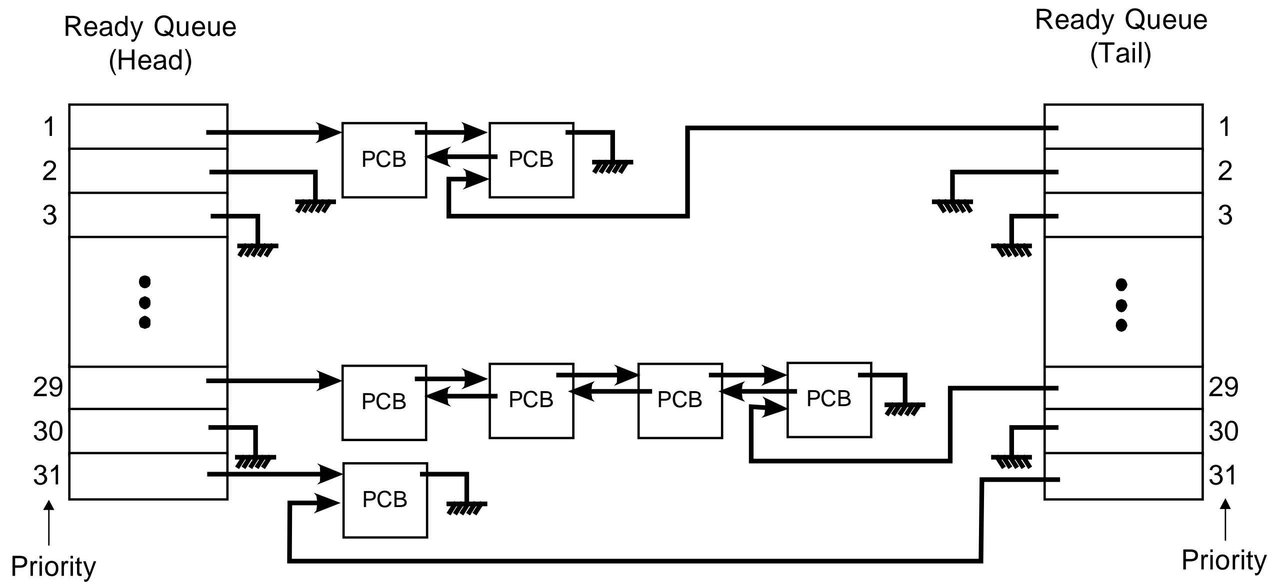
Each P in the system is represented by a PCB. PCBs are members of a P queue.
Each P state has associated one or several queues. Running state has a single one-member queue. Blocked state has several queues, one for each reason of blockage (devices, synchronization objects.) Switching of a P from one state to another state results in relinking of the corresponding PCB from one queue to another.



Copyright © 1998-2002 by Marko Vuskovic

PRIORITIZED READY QUEUE

In order to optimize the behavior of the system the ready queue is usually prioritized.
For that purpose a priority number is assigned to each P. Consequently there will be as many ready queues as there are priority numbers. (For example, Windows NT has 31 priorities which range from 1 to 31, while UNIX has 128 priorites which range from 0 - high to 127 -low.)



Copyright © 1998-2002 by Marko Vuskovic

PROCESS SWITCHING

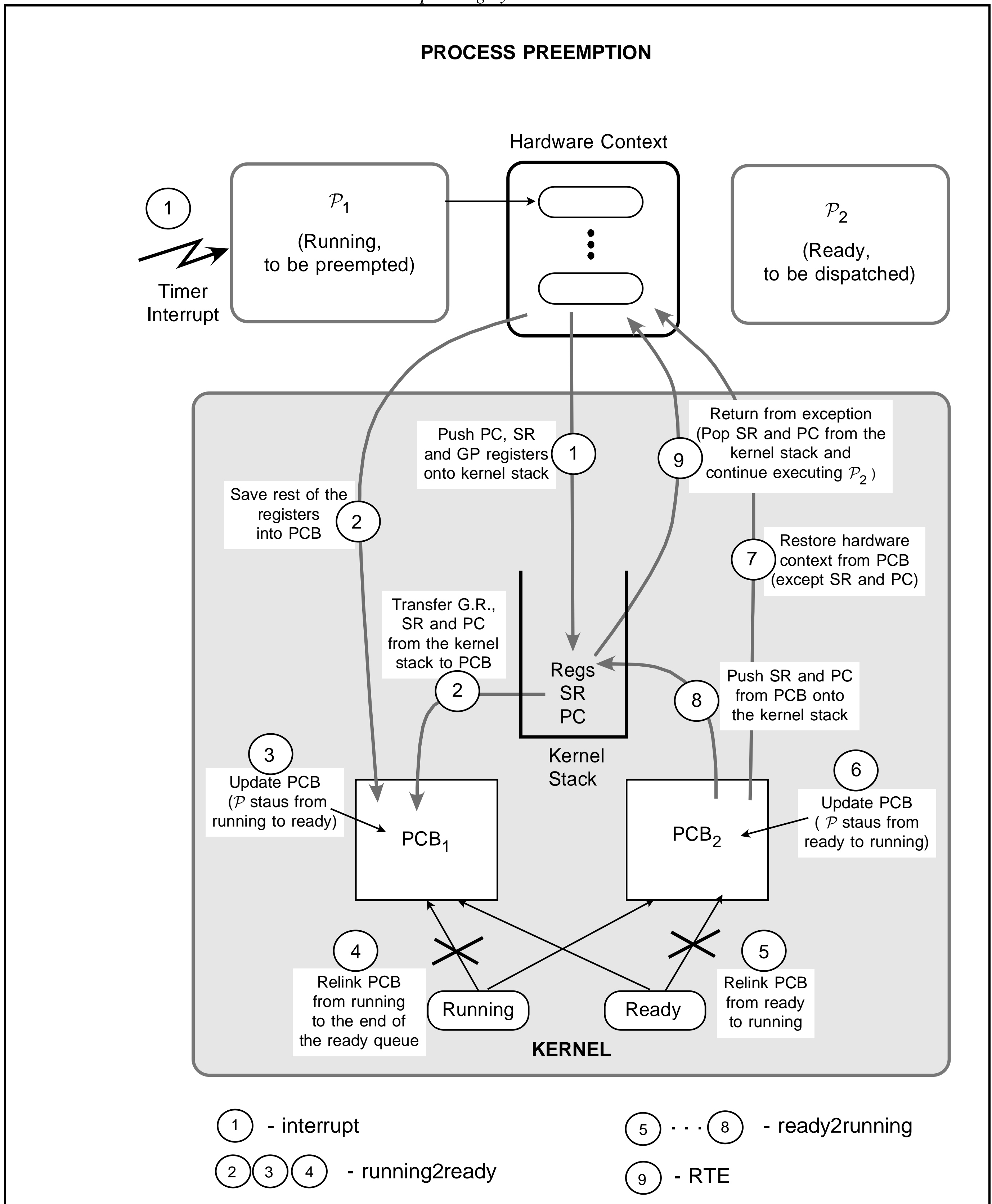
The following events trigger transition of a $\mathcal{P}$ from one state to another:

1. Time quantum (time slice) of the running $\mathcal{P}$ has expired - Timer interrupt
2. Memory fault - MMU interrupt
3. I/O completion - I/O interrupt
4. Deblocking condition has occurred (due to a synchronization object) - System call
5. $\mathcal{P}$ enters blocked state due to an I/O request or synchronization object - System call
6. $\mathcal{P}$ terminates execution - System call
7. New $\mathcal{P}$ has entered the system - System call
8. Priority of some $\mathcal{P}$ has changed - System call

Notice that all these events are causing an exception, either a hardware interrupt or a software trap. Therefore the process switching is performed from within an exception handler.

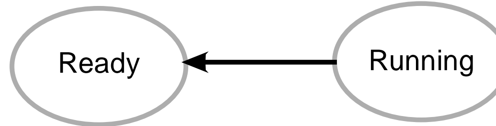
On the following pages will be discussed cases 1, 3, 4 and 5. Other cases are similar to these cases.

Copyright © 1998-2002 by Marko Vuskovic



Copyright © 1998-2002 by Marko Vuskovic

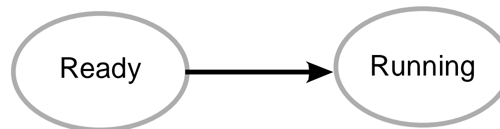
DISPATCHER PRIMITIVES



Preempting

```
running2ready()   // Move running process to ready queue
{                 //  (PCB denotes process control block
                  //  of the running process)

   ① Transfer g.p. registers from the kernel stack to PCB;
   ② Transfer SR and PC from the kernel stack to PCB;
   ② Save rest of the hardware context to PCB;
   ③ Update PCB (process state -> ready);
   ④ Relink PCB from running to the end of the ready queue;
}
```



Dispatching

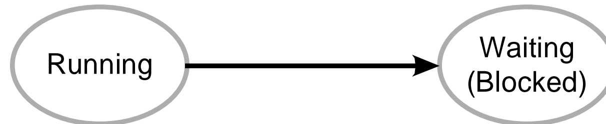
```
ready2running()  // Move selected proc. from ready to running queue
{                //  (PCB denotes process control block
                 //  of the selected process)

   ⑤ Relink PCB from ready to running queue¹;
   ⑥ Update PCB (process state -> running);
   ⑦ Restore hardware context (except SR and PC)from PCB;
   ⑧ Push SR and PC onto kernel stack;
}
```

[1]) If the ready queue is prioritized, i.e. if there is a separate queue for each priority, typically the PCB at the head of the queue would be selected. In that case macro "ready2running()" doesn't need argument that specifies the queue element. In addition, such queues don't need to be doubly linked.

Copyright © 1998-2002 by Marko Vuskovic

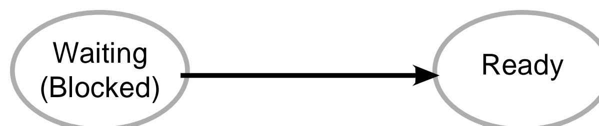
DISPATCHER PRIMITIVES (Cont.)



Blocking

```
running2blocked(h)      // Move running process to blocked queue
{                       // with handle h
                        // (PCB denotes process control block
                        //  of the running process)

     Transfer g.p. registers from the kernel stack to PCB;
     Transfer SR and PC from the kernel stack to PCB;
     Save rest of the hardware context to PCB;
     Update PCB (process state -> blocked);
     Relink PCB from the running to the end of the blocked
      queue with handle h;
}
```



Unblocking

```
blocked2ready(h)        // Move all processes from blocked queue h
{                       // to ready queue

     Relink PCB from blocked queue to the end of the ready queue;
     Update PCB (process state -> ready);
}
```

Copyright © 1998-2002 by Marko Vuskovic

TIME QUANTUM

```
{     // This is the timer interrupt handler, therefore SR and PC
      // of the running process were pushed onto kernel stack

      Disable interrupts;
      Save g.p. registers onto kernel stack;
      Update system time;
      Decrement time quota of the running process (in PCB);
      if (time quota = 0)
      {                                    <- Preempting
          running2ready();     // Move running process to ready queue
          ready2running();     // Move next proces to running queue
      }                                    <- Dispatching
      else
      {
          restore g.p. registers;
      }
      RTE;                     // Return from handler
                               // (There is no need to enable interrupts
                               // since RTE loads SR of the user process
                               // or thread, which has interrupts enabled)
}
```


COMMENTS:

The algorithm above corresponds to the Round Robin scheduling discussed later.

Operation "running2ready()" is called process preemption.

Operation ready2running() is called process dispatching.

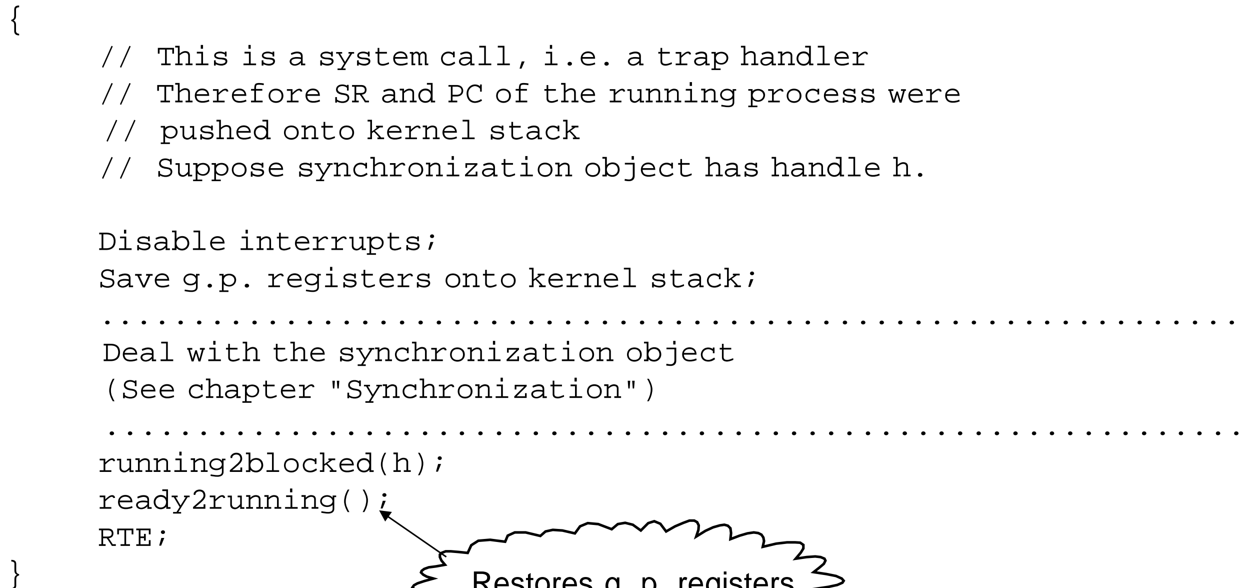
Sequence: running2ready(); ready2running(); is called process switching, which implies context switching.

The algorithms above (and on the following pages) are typical. However, Windows NT uses more complex algorithms which employ deferred procedure call. These will be discussed later in the case study.

Process switching must be uninterrupted operation, therefore the interrupts are disabled at the beginning. Since the process switching is an operation with generally a finite and short duration, disabling interrupts is not too dangerous.

Copyright © 1998-2002 by Marko Vuskovic

BLOCKING EVENT



```
{
      // This is a system call, i.e. a trap handler
      // Therefore SR and PC of the running process were
      // pushed onto kernel stack
      // Suppose synchronization object has handle h.

      Disable interrupts;
      Save g.p. registers onto kernel stack;
      ............................................................
      Deal with the synchronization object
      (See chapter "Synchronization")
      ............................................................
      running2blocked(h);
      ready2running();
      RTE;
}
```


NOTICE: There is no need to restore general purpose registers from the kernel stack since the registers of the new running process are restored by the macro *ready2running()*. The registers pushed onto stack (which belong to the process that is making this system call) are popped by the macro *running2blocked()*.

Copyright © 1998-2002 by Marko Vuskovic

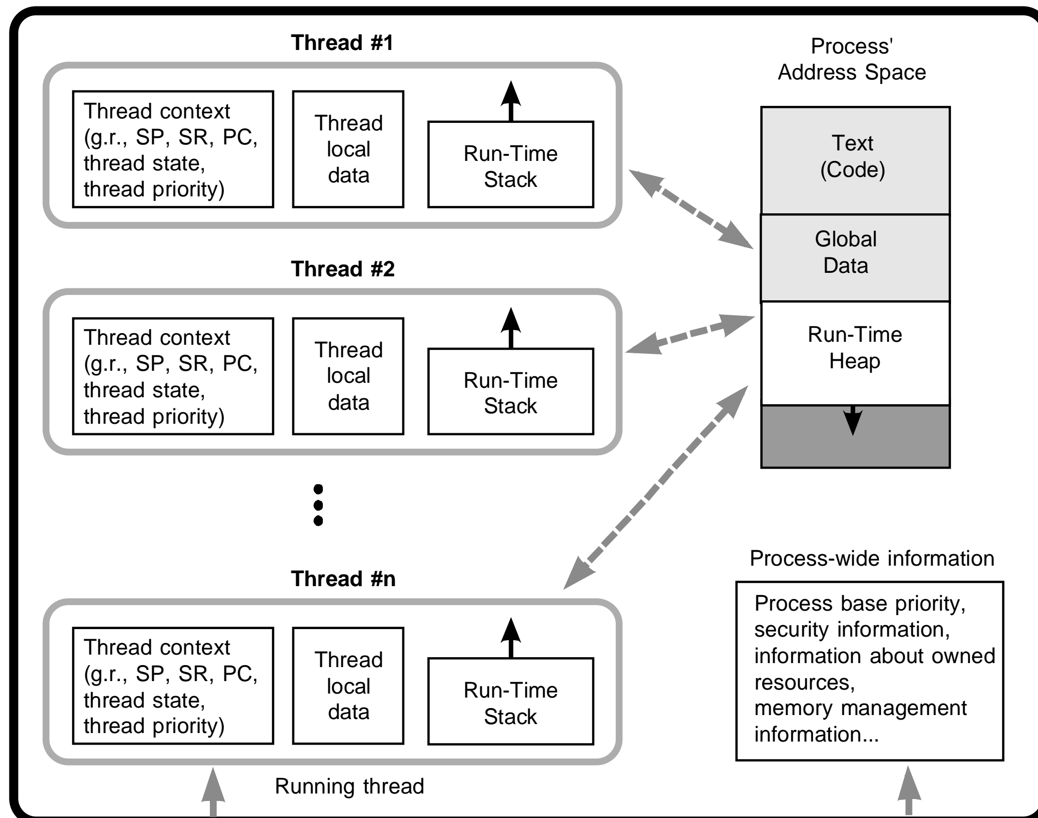
I/O INTERRUPT OR DEBLOCKING EVENT

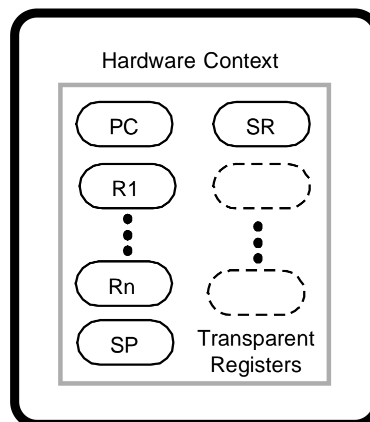
```
{
      // This is an interrupt handler (in case of I/O completion)
      // or a trap handler (in case of a deblocking system call).
      // Therefore SR and PC of the running process were
      // pushed onto kernel stack
      // Suppose the blocked queue associated with the interrupting
      // I/O device or synchronization object has handle h.

      Disable interrupts;
      Save g.p. registers onto kernel stack;
      .............................................................
      Transfer data (I/O interrupt) or deal with the synchronization
      object (see chapter "Synchronization")
      .............................................................
      while (blocked queue h not empty)
      {
            blocked2ready(h);
      }
      if (the highest priority in ready queue is higher
         than priority of the running process)
      {
            running2ready();
            ready2running();
      }
      else
      {
            restore g.p. registers;
      }
      RTE;                         // Return from handler
}
```

Copyright © 1998-2002 by Marko Vuskovic

STRUCTURE OF THREADS





Copyright © 1998-2002 by Marko Vuskovic

THREADS

Threads of execution, or abbreviated "threads" (𝒯) have smaller context and therefore faster context switching than processes. For that reason, 𝒯s are sometimes called "lightweight processes." A process can have one or any number of 𝒯s.

Second important fact is that 𝒯s share the address space and resources with the 𝒫 which they belong to. Consequently, all 𝒯s in a 𝒫 can access 𝒫's files, global data and run-time heap. Since the resources and data the 𝒯s are accessing are within the same protection shell, there is no need to do the access via system calls. This is important in applications which require several cooperating and concurrent activities (see examples below).

Multiple 𝒯s allow a program to be divided into tasks that can operate independently from each other.𝒯s are not always required to achieve this result but they can make the programmer's job easier.

Ts were introduced in VMS and Mach OS and are available now in OS/2, Windows 95, Windows NT, Solaris and HPUX. (Mach is an evolutionary extension of UNIX and is the basis for Open Software Foundation.)

In Windows NT 𝒫s are passive and take care of address space, resources and security, while 𝒯s are active and carry the execution. Therefore, it is commonly said: 𝒯 is a unit of dispatching, while 𝒫 is a unit of resource ownership.

In Windows NT each 𝒫 has at least one 𝒯, called primary thread, because process cannot be scheduled. This 𝒯 is created automatically upon the creation of the 𝒫 (*CreateProcess()*). Additional 𝒯s are created explicitly by a system call (*CreateThread()*)

𝒯 switching (for example between running and ready state) is very fast if 𝒯s belong to the same 𝒫. Otherwise, 𝒯 switching requires also 𝒫 switching, which involves longer context switching.

Two 𝒯s can exchange data via global variables or via run-time heap, only if they belong to the same 𝒫. Ts from different 𝒫s cannot communicate directly because they generally execute in different address spaces. Therefore special communication mechanisms must be established, which are called Inter Process Communication (IPC), see chapter about IPC.

If two Ts from the same P access the same global data, the access must be serialized in order to maintain data integrity. Mechanisms for serialization of data access is discussed in chapter "Synchronization."

NOTICE: The P states and process switching discussed in previous pages applies to Ts in a treaded operating system. From now on we will suppose such system.

Copyright © 1998-2002 by Marko Vuskovic

THREAD EXAMPLE

```
//*********************************************************************
// Thread_Example.cpp
//
//        This program demonstrates threads.
//        The main program (the primary thread) creates two
//        threads. Each thread announces its birth, then it
//        increments a global variable. When the value of the
//        global variable exceeds 10, threads terminate.
//        After creating threads, the primary thread enters
//        a loop in which it prints the value of the global variable
//        every 200 milliseconds. The loop is terminated when both
//        threads die.
//********************************************************************

#include <windows.h>
#include <iostream.h>

int c = 0;                    // Global variable
                              // (Can be seen by threads)
// Thread function (defines the thread behavior)
void fun(DWORD i)
{
     cout << "============ Thread #" << i
          << ": I am just born!"
          << endl;

     while (c < 10)
     {
          c++;               //Increment global the variable
          Sleep(1000);       // Sleep 1 second
     }
     cout << "============ Thread #"<< i
          << ": Bye, I am going away now"
          << endl;
}

// The primary thread
void main()
{
    HANDLE t[2];             // Thread handles
    DWORD threadID[2];       // Thread ID (returned by thread)

    cout << "Main: Hi, I am the primary thread" << endl;
    Sleep(2000);
```

Copyright © 1998-2002 by Marko Vuskovic

```
    cout << "Main: I am creating two threads now" << endl;

    // Create thread #1:
    t[0] = CreateThread(
            0,                  //Default security parameters
            0,                  //Default thread stack size
            (LPTHREAD_START_ROUTINE) fun,
            (void *)1,          // Parameter passed to thread
            0,                  // Default creation flags
            &threadID[0]);
    if(t[0] == INVALID_HANDLE_VALUE)
    {
        cout << "Main: Can't create thread #1, Error code = "
             << GetLastError() <<endl;
        CloseHandle(t[0]);
    }
    cout << "Main: Thread #1 has ID = " << threadID[0]
         << ", and handle = " << t[0] << endl;


    // Create thread #2:
    t[1] = CreateThread(
            0,                  //Default security parameters
            0,                  //Default thread stack size
            (LPTHREAD_START_ROUTINE) fun,
            (void *)2,          // Parameter passed to thread
            0,                  // Default creation flags
            &threadID[1]);
    if(t[1] == INVALID_HANDLE_VALUE)
    {
        cout << "Main: Can't create thread #2, Error code = "
             << GetLastError() <<endl;
        CloseHandle(t[1]);
    }


    cout << "Main: Thread #2 has ID = " << threadID[1]
        << ", and handle = " << t[1] << endl;

    cout << "Main: Now the treads are created, they will soon
report"
         << endl <<endl
```

Copyright © 1998-2002 by Marko Vuskovic

```
    // Main thread loop
    // (this loop exits when both threads, #1 and #2 exit)

    while (WaitForMultipleObjects(
                  2,       // Two objects
                  t,       // Array of handles
                  TRUE,    // Wait for both objects to be signaled
                  0        // Time-out interval (exit immediately)
          ) == WAIT_TIMEOUT)
    {
         cout << "Main: c = " << c << endl;
         Sleep(200);          // Report every 200 milliseconds
    }


    cout << "Main: By now both threads have died, and I am going to
exit too, bye"
         << endl;
```

Copyright © 1998-2002 by Marko Vuskovic

Manual Page for CreateThread() System Function

CreateThread

The **CreateThread** function creates a thread to execute within the address space of the calling process.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  // pointer to security attributes
  DWORD dwStackSize,                         // initial thread stack size
  LPTHREAD_START_ROUTINE lpStartAddress,     // pointer to thread function
  LPVOID lpParameter,                        // argument for new thread
  DWORD dwCreationFlags,                     // creation flags
  LPDWORD lpThreadId                         // pointer to receive thread ID
);
```

Parameters

lpThreadAttributes

Pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

Windows NT: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new thread. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor.

dwStackSize

Specifies the initial commit size of the stack, in bytes. The system rounds this value to the nearest page. If this value is zero, or is smaller than the default commit size, the default is to use the same size as the calling thread. For more information, see Thread Stack Size.

lpStartAddress

Pointer to the application-defined function of type LPTHREAD_START_ROUTINE to be executed by the thread and represents the starting address of the thread. For more information on the thread function, see ThreadProc.

lpParameter

Specifies a single 32-bit parameter value passed to the thread.

dwCreationFlags

Specifies additional flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state, and will not run until the ResumkeThread function is called. If this value is zero, the thread runs immediately after creation. At this time, no other values are supported.

lpThreadId

Pointer to a 32-bit variable that receives the thread identifier.

Windows NT: If this parameter is NULL, the thread identifier is not returned.

Windows 95 and Windows 98: This parameter may not be NULL.

Copyright © 1998-2002 by Marko Vuskovic

Manual Page for CreateThread() System Function (Cont.)

Return Values

If the function succeeds, the return value is a handle to the new thread.
If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Windows 95 and Windows 98: CreateThread succeeds only when it is called in the context of a 32-bit program. A 32-bit DLL cannot create an additional thread when that DLL is being called by a 16-bit program.

Remarks

The new thread handle is created with THREAD_ALL_ACCESS to the new thread. If a security descriptor is not provided, the handle can be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread.
The thread execution begins at the function specified by the *lpStartAddress* parameter. If this function returns, the **DWORD** return value is used to terminate the thread in an implicit call to the ExitThread function. Use the GetExitCodeThread function to get the thread's return value.
The **CreateThread** function may succeed even if *lpStartAddress* points to data, code, or is not accessible. If the start address is invalid when the thread runs, an exception occurs, and the thread terminates. Thread termination due to a invalid start address is handled as an error exit for the thread's process. This behavior is similar to the asynchronous nature of **CreateProcess**, where the process is created even if it refers to invalid or missing dynamic-link libraries (DLLs).
The thread is created with a thread priority of THREAD_PRIORITY_NORMAL. Use the GetThreadPriority and SetThreadPriority functions to get and set the priority value of a thread.
When a thread terminates, the thread object attains a signaled state, satisfying any threads that were waiting on the object.
The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to CloseHandle.

Copyright © 1998-2002 by Marko Vuskovic

Manual Page for CreateThread() System Function (Cont.)

The ExitProcess, ExitThread, **CreateThread**, CreateRemoteThread functions, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means that the following restrictions hold:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.
- Only one thread in a process can be in a DLL initialization or detach routine at a time.
- **ExitProcess** does not return until no threads are in their DLL initialization or detach routines.

A thread that uses functions from the C run-time libraries should use the **beginthread** and **endthread** C run-time functions for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when **ExitThread** is called.
Windows CE: The *lpThreadAttributes* parameter must be set to NULL. The *dwStackSize* parameter must be zero. Only zero or CREATE_SUSPENDED values are supported for the *dwCreationFlags* parameter.

QuickInfo

Windows NT: Requires version 3.1 or later.
Windows: Requires Windows 95 or later.
Windows CE: Requires version 1.01 or later.
Header: Declared in winbase.h.
Import Library: Use kernel32.lib.

Copyright © 1998-2002 by Marko Vuskovic

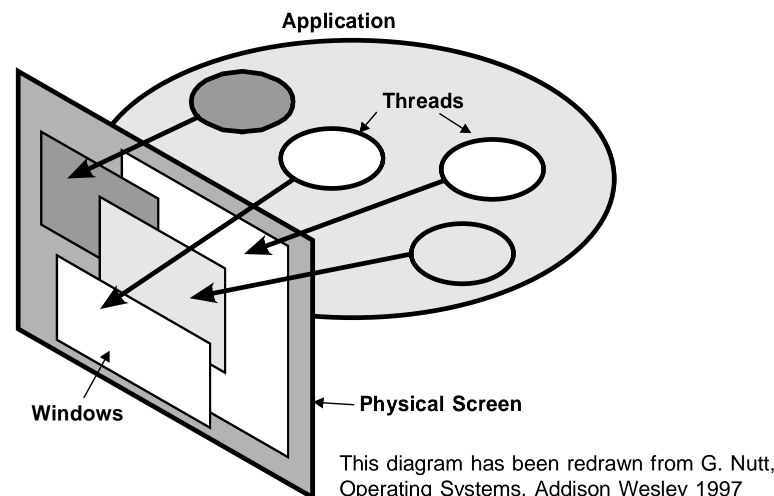
EXAMPLES OF USAGE OF THREADS

Network servers (File servers, Web servers, ticket reservation systems...) Server is normally a single process which performs few operations on common data (global data, files) or I/O devices (laser printer). Each concurent request from a client would create a T and after finishing the job the T would be destroyed. The full benefit of threading one has if the server runs on several processors. However, even on a single processor, threads simplify server design and implementation.

Embedded systems (Autopilots, appliences, elevators, robotic systems, real-time simulators, factory process control...) They conist of several concurrent and cooperative activities. The time efficiency is there extremely important, therefore fast context switching and direct access of data with minimal support of the operating system executive is crucial.

Multiprocessoring (Application which run on several CPUs). The application is split into multiple Ts which then can execute simultaneously, usually working on the same data (robotics is a good example.)

Graphical User Interface (GUI) Some applications may have several windows on the screen and therefore several points of user interaction. Each of these windows can be controlled by a separate Ts. In addition to handling user inputs, there are background activities like refreshing the graphics, periodic saving of data from RAM to disk file, garbage collection and compaction, etc. Multiple Ts make the application more responsive since none of the interactive inpust will hang for longer time if some activity takes a longer time to complete. (Compare Windows File Manager and the new MS Explorer. Multiple Document Interface (MDI) used in word processors is another example.)



This diagram has been redrawn from G. Nutt, Operating Systems, Addison Wesley 1997

Where not to use threads:

- A segment of code that has to be synchronized with the application (for example the application is waiting for data before it can continue execution) should not be a thread. Making such segment of code a thread would require very complex thread synchronization with a very little benefit.

Copyright © 1998-2002 by Marko Vuskovic

THREAD SCHEDULING

In the first half of this chapter we have discussed processes and threads, their states, their execution environment and their structure from the programming point of view. In the rest of the chapter we will discuss their scheduling. As a matter of fact, in the pseudocode for the kernel routines that handle time quantum, blocking events and I/O interrupts (see pages 3-16, 17, 18) we have assumed that the process scheduling is implicitly done somehow:

```
running2ready();     // Move running process to ready queue
ready2running();     // Move next process to running queue
```

Now we will see exactly how the next process is chosen to be put into the running queue.

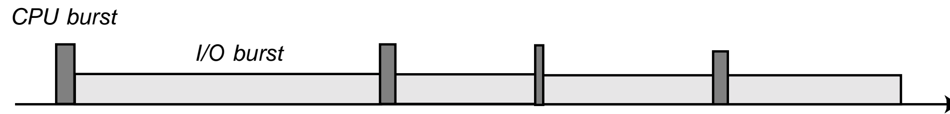
NOTICE: In the further discussion we will assume threaded operating systems, therefore we will be talking about thread scheduling (not process scheduling). In other words, threads will be carrying the execution and will be subject to scheduling, while the processes will be responsible for resources necessary for threads to work.

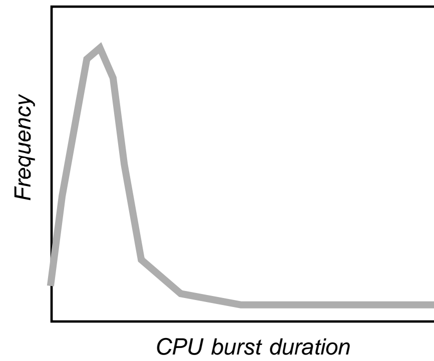
Copyright © 1998-2002 by Marko Vuskovic

CPU AND I/O BURSTS

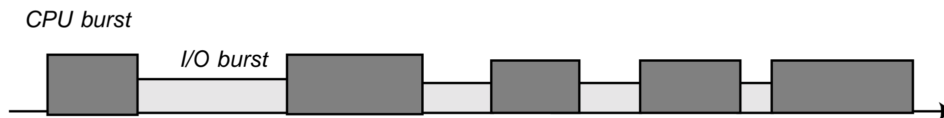
A thread execution alternates between two general modes: CPU burst and I/O burst. CPU bursts is when the thread is executing on CPU, while I/O burst is when thread is blocked and waiting for I/O transfer.

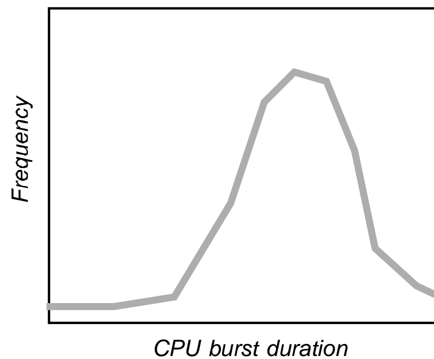
I/O-bound thread:





CPU-bound thread:





Copyright © 1998-2002 by Marko Vuskovic

SCHEDULER CRITERIA

CPU Utilization

Percentage of time that the CPU is busy
Important in expensive shared systems (super computers), less important in single user and personal systems, not important in real-time systems.

Throughput

Number of jobs processed per unit of time.

Turnaround Time

Time between the submission of the job and its completion.
(Includes: CPU time and time spent waiting for resources.)

Waiting Time

Amount of time the job has spent in ready queue (i.e. its threads)

Response Time

Time between the submission of a request (not job!) and the first response.
(Requests and results can be "pipelined": new results can be still processed while the previous results are displayed.)
Better criteria than the Turnaround Time for interactive jobs.

SCHEDULING METHODS:

- First Come, First Served (FCFS)
- Shortest Job First (SJF)
- Priority Scheduling
- Round-Robin Scheduling
- Multiprocessor Scheduling

Copyright © 1998-2002 by Marko Vuskovic

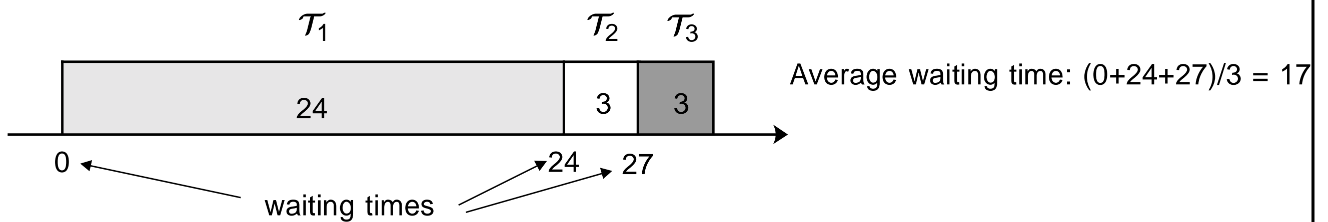
FIRST-COME, FIRST-SERVED SCHEDULING (FCFS)

The simplest CPU scheduling algorithm.

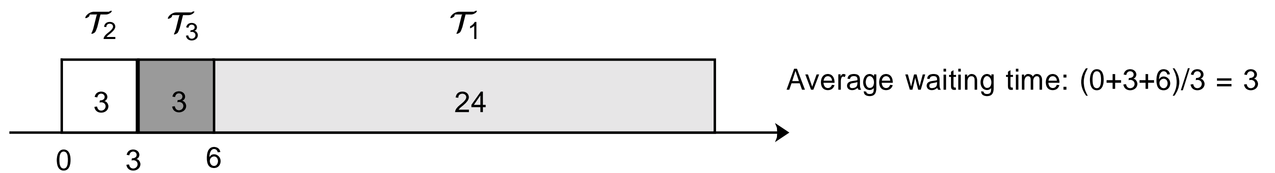
Nonpreemptive scheduling algorithm.

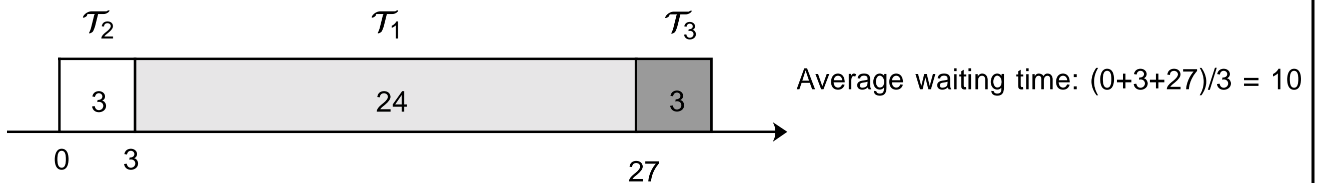
Ready queue is used as a FIFO queue: Ready $\mathcal{T}$s enter the tail of the queue, CPU is allocated to the $\mathcal{T}$ at the head of the queue.

Long average waiting time:



Different order:





(Simplified analysis: only one burst time per thread is considered here. More accurate analysis requires simulation models.)

The variations of average waiting time can be very large in mixes with large variations of CPU-burst times.

Convoy effect: Many short I/O-bound $\mathcal{T}$s wait one big CPU-bound $\mathcal{T}$ to get off the CPU.

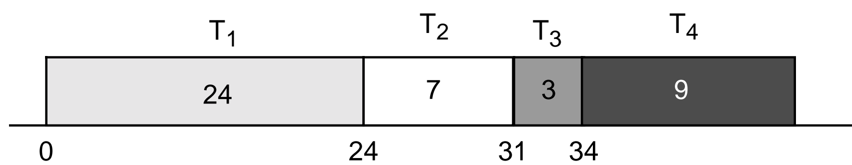
Copyright © 1998-2002 by Marko Vuskovic

SHORTEST-JOB-FIRST SCHEDULING (SJF)

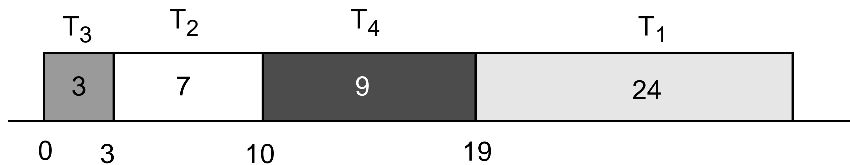
More precise name would be: Shortest-Next-Burst-First, because the length of the next CPU-burst is examined instead of the total length of the job.

Generally nonpreemptive scheduling algorithm.

Provides minimal average waiting time.



Average waiting time: (0+24+31+34)/4 = 22.25



Average waiting time: (0+3+10+19)/4 = 8

Problem: How to know the size of the next CPU-burts time of each T ?.

Therefore a prediction can be made based on the past CPU-bursts.
Well known algorithm for prediction is the exponential average of CPU-burst times, which is maintained for each thread:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\, \tau_n$$

where:

- τ_n - predicted burst time in the previous cycle
- τ_{n+1} - predicted burst time in the next cycle
- t_n - measured burst time in the previous cycle
- α - prediction parameter (small α puts more weight to the history)

Extreme cases:

- $\alpha = 0$, then $\tau_{n+1} = \tau_n$ meaning that algorithm uses a constant value
- $\alpha = 1$, then $\tau_{n+1} = t_n$ - only recent measured burst times matter

Copyright © 1998-2002 by Marko Vuskovic

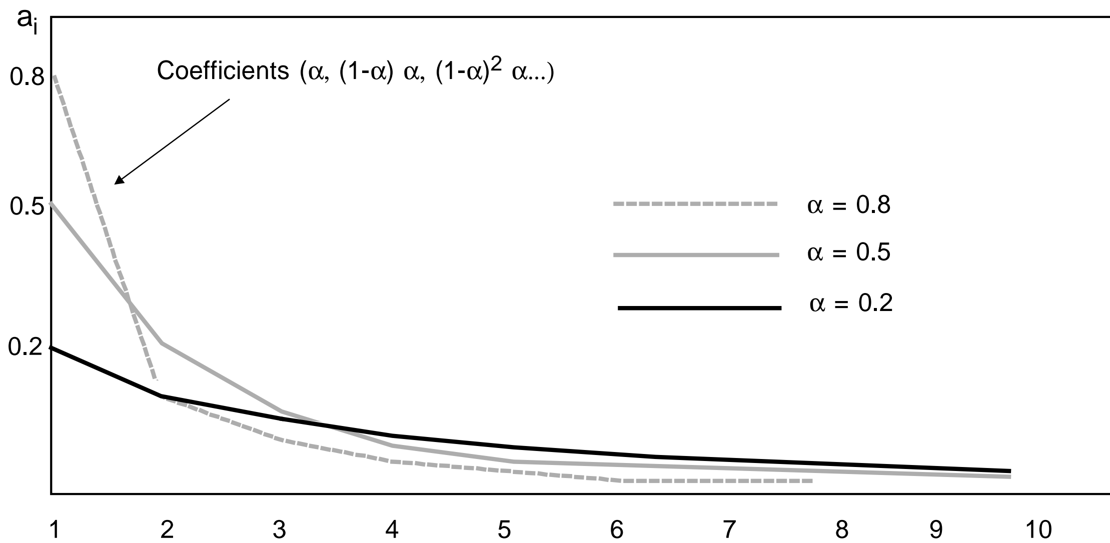
Behavior of the exponential average can be obtained by successive substitutions of recurrence equation on the previous page:

$$\tau_{n+1} = a_1\, t_n + a_2\, t_{n-1} + a_3\, t_{n-2} + \ldots + a_n\, t_1 + (1-\alpha)^{n-1}\tau_1$$

where:

$$a_i = \alpha\,(1-\alpha)^{i-1}$$

(τ_1 can be approximated by t_1)



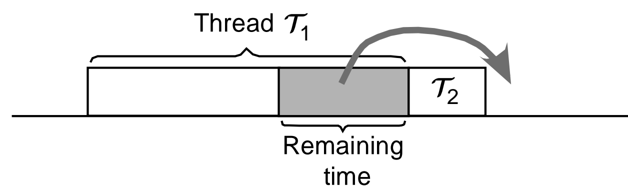
The older the observation, the less is counted into the average.

Larger values for α quickly reflect rapid changes in burst times.
However the short surges can make the behavior "jerky".

Possibility of starvation for jobs with large CPU-burst times,
if there is a steady supply of shorter jobs.

Shortest-remaining-time-first algorithm:

Preemptive version of SJF algorithm.
If a new $\mathcal{T}$ with the burst time shorter than the remaining time of the currently running $\mathcal{T}$ comes into ready queue it will be preempted.



Copyright © 1998-2002 by Marko Vuskovic

PRIORITY SCHEDULING

Each $\mathcal{T}$ is given a priority. (Algorithm similar to SJF, if priority equals to the inverse of the next burst time)

$\mathcal{T}$s with equal priority are scheduled by FCFS.

Priorities can be determined internally by the operating system which can map various thread attributes like time limits, memory requirements, the number of open files, ratio of average I/O to CPU bursts.

Priorities can also be determined externally by operator or system manager in accordance with the policy, financial factors or importance of the job/customer.

Priority scheduling can be nonpreemptive or preemptive. In preemptive case, the running $\mathcal{T}$ will be preempted if the new $\mathcal{T}$ which enters the ready queue has higher priority. The preempted $\mathcal{T}$ is put at the head of ready queue.

Like with SJF, priority algorithm can suffer from starvation - low priority $\mathcal{T}$s can wait indefinitely for the CPU (case at MIT in 1973)

Aging, technique which gradually increases the priority of long-waiting, low-priority $\mathcal{T}$s - used to prevent the starvation of low-priority $\mathcal{T}$s. For example, the priority can be incremented by some value every 15 minutes.

Copyright © 1998-2002 by Marko Vuskovic

ROUND-ROBIN SCHEDULING

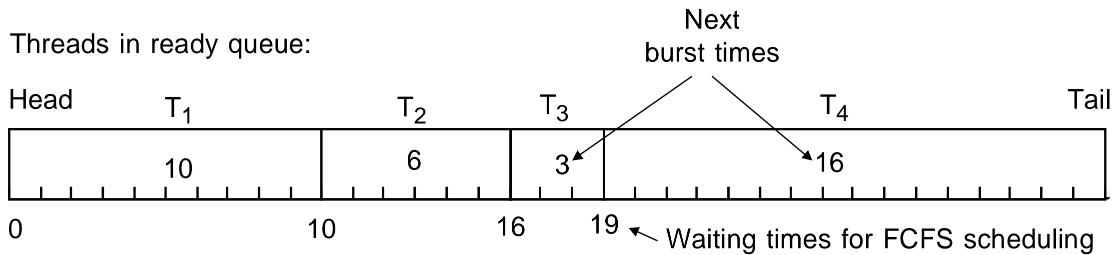
Designed for time-sharing systems.

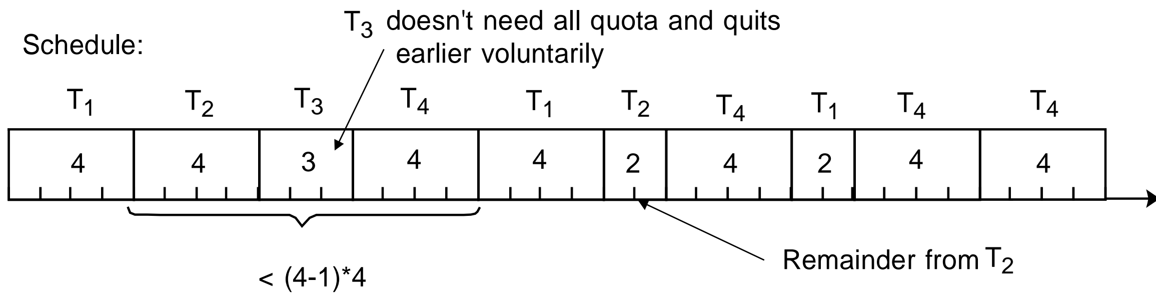
Each T is given a time quantum, or time slice (usually 10 to 100 ms, NT typically 20 ms)

After T has used its time quantum it will be preempted. The preempted T is inserted at the tail, while the next T is taken from the head of the ready queue. In other words, access to the ready queue is FIFO.

Example (time quantum: q = 4):

Threads in ready queue:





Waiting times:

		RR	FCFS	Waiting for the next time quantum (RR)
T_1	= 4+3+4+2+4 =	17	0	11, 6
T_2	= 4+3+4+4 =	15	10	11
T_3	= 4+4 =	8	16	0
T_4	= 4+4+3+4+2+2 =	19	19	6, 2, 0
	Average:	14.75	11.25	

(Waiting for the next time quantum: all < 12)

Each T must wait no longer than (n-1)*q time units for the next time quantum

Copyright © 1998-2002 by Marko Vuskovic

ROUND-ROBIN SCHEDULING (Cont.)

Performanse of RR scheduling depends largely on the size of the time quantum (q).

Very large q (larger than maximal CPU-burst time) -- RR is equivalent to FCFS

Very small q (one time unit) -- too much overhead for contex switching.

Rule of thumb: choose quantum which fits typical CPU-burst times for I/O bound $\mathcal{T}$s.

In a mix of I/O-bound and CPU-bound threads, the threads which do not have to block often have advantage over threads that block and unblock often. There are two approaches to increase the fairness of RR, i.e. to allow the I/O-bound $\mathcal{T}$s to compete with the CPU-bound $\mathcal{T}$s:

Virtual Round Robin (VRR)

Blocked $\mathcal{T}$s go after deblocking into a special auxilliary queue instead of ready queue, while preempted $\mathcal{T}$s go into ready queue. The dispatcher would first look into the auxilliary queue, then into the ready queue.

Variable priorities (used in Windows NT)

Uses prioritized ready queue. Each preempted $\mathcal{T}$ is gradually decreased the priority, while each deblocked $\mathcal{T}$ is gradually increased the priority. This is called: priority boosting.

This however can cause another problem: if there are $\mathcal{T}$s which block and deblock very often, they will soon increase the priority way above their base value, which will cause their monopolization of the CPU. In order to correct this condition, the deblocked $\mathcal{T}$s retain their cumulative time quota (as shown on page 3-28). Consequently, the I/O-bound $\mathcal{T}$s will eventually exhaust their quota and will be preempted. When this happens, the scheduler will lower their priority back to the base value, and the cycle of the priority boost repeats.

In prioritized RR there is another problem called priority inversion. Suppose there are three $\mathcal{T}$s: one with low priority ($\mathcal{T}_1$), one with medium priority ($\mathcal{T}_2$), and one with high priority ($\mathcal{T}_3$). Also suppose that $\mathcal{T}_1$ holds some resource. Now if $\mathcal{T}_3$ gets blocked on the same resource $\mathcal{T}_1$ is holding, the high priority thread $\mathcal{T}_3$ will never get the resource because $\mathcal{T}_1$ will never get the chance to release the resource due to $\mathcal{T}_2$ which has higher priority. In order to prevent this priority inversion condition, the scheduler boosts the priority of the low priority thread that owns a resource to the level of threads that are blocked on the same resource.

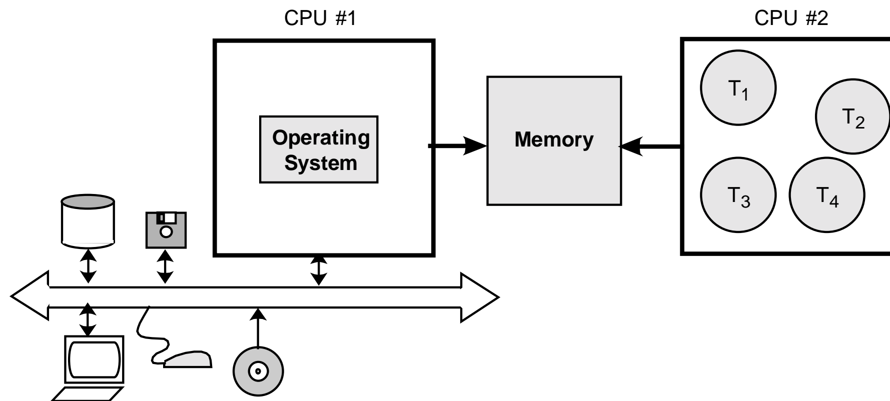
There are other scheduling algorithms discussed in various OS books, like Multilevel Queue Scheduling and Multilevel Feedback Queue Scheduling, which are essentialy RR with fixed and variable priorities.

Copyright © 1998-2002 by Marko Vuskovic

MULTIPROCESSOR SCHEDULING

There are generally two types of tightly-coupled multiprocessors:

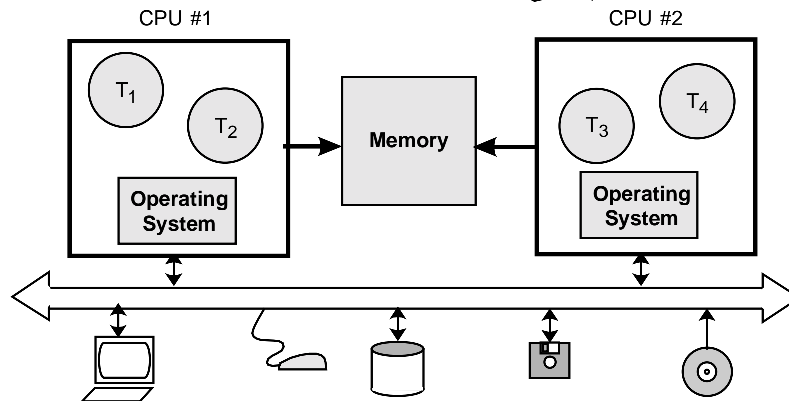
Asymetric Multiprocessing (AMP)



Simpler to implement.
One processor is dedicated to the operating system, while other processor(s) is dedicated to run the user threads.

Symetric Multiprocessing (SMP)

Used by NT



Better utilization of the resources and better balancing of the system load.
Fault tolerance: if one CPU fails, system would still be working, because the OS can continue to work on another CPU.
Synchronization more complicated than with the uniprocessor machines.

Scheduling: One ready queue, several running "queues".
All scheduling algorithms described earlier can be applied here.

Copyright © 1998-2002 by Marko Vuskovic

SCHEDULER HIERARCHY

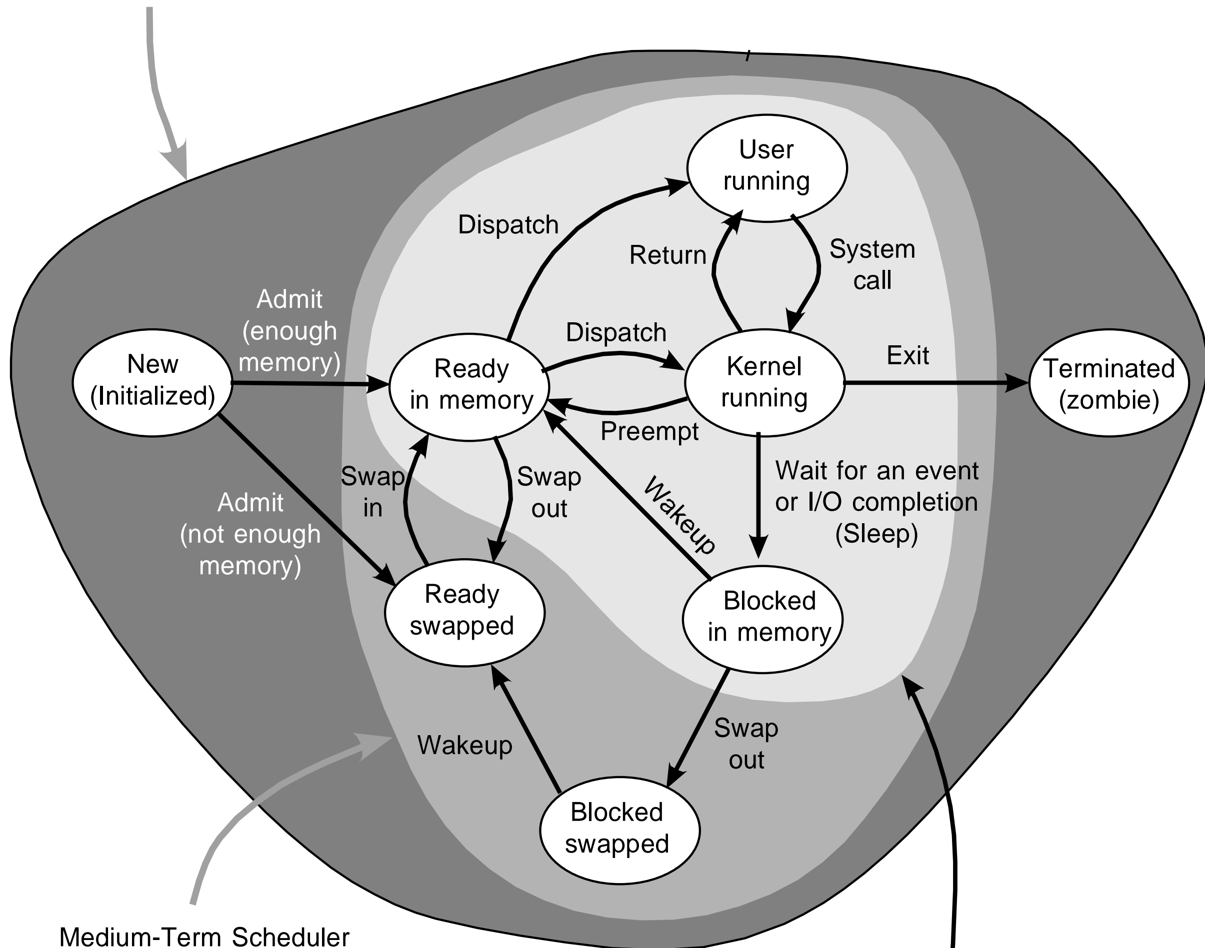
Long-Term scheduling

Decide whether to admit a new job (limit the degree of multiprogramming, $\Delta\tau = T/n$)

Decide which job to admit (consider: I/O requirements, expected execution time, priorities, desired mix: CPU λ + I/O (1-λ)

Related to resource allocation.

Specially important in systems with mix of batch jobs and interactive users.



Medium-Term Scheduler

Part of the swapping function.

Considers amount of required memory.

Short-Term Scheduler

Also called: Low-Level Scheduler

Also called: Dispatcher

(will be discussed in the following sections.)

Copyright © 1998-2002 by Marko Vuskovic