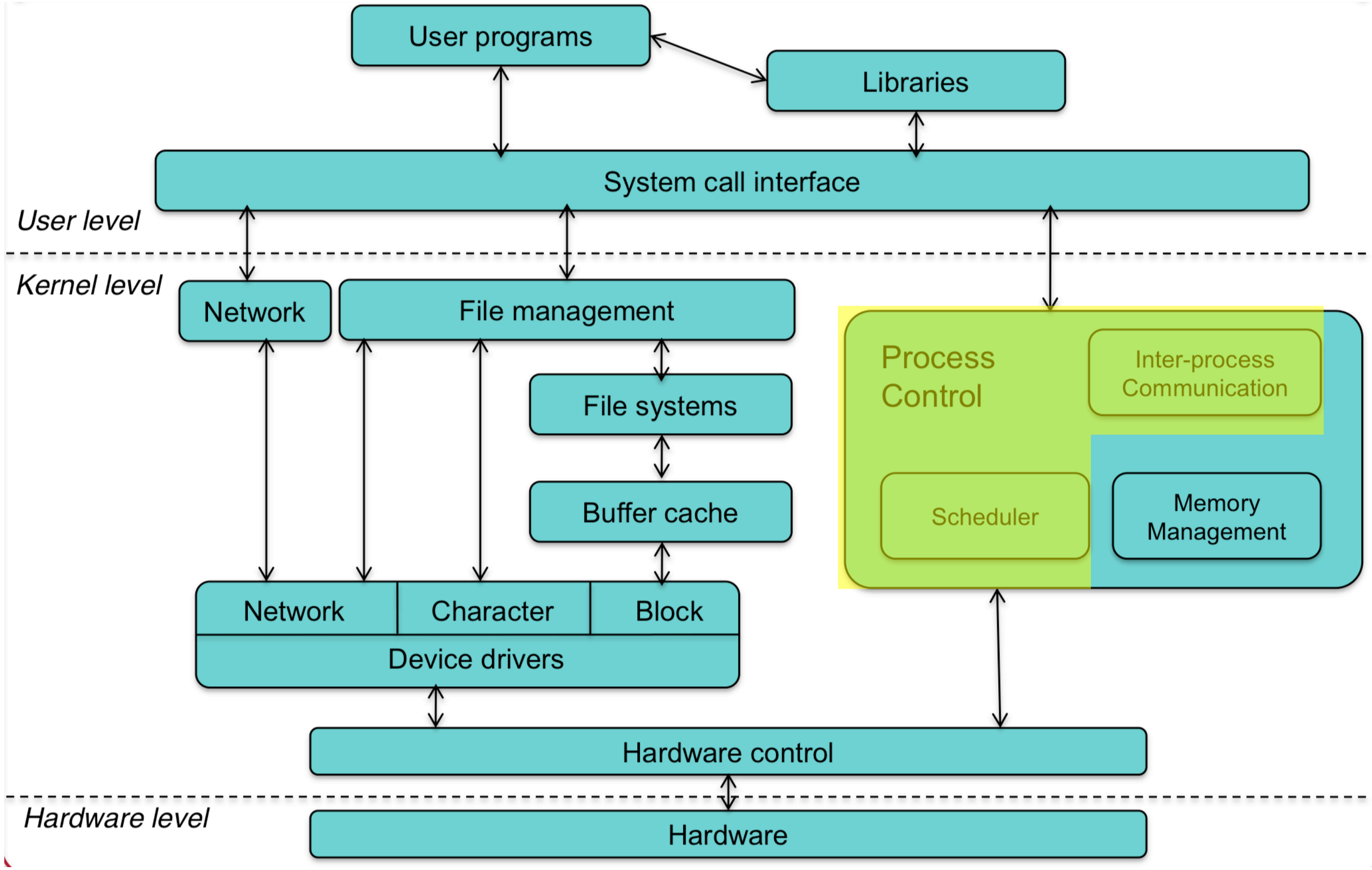
Operating Systems

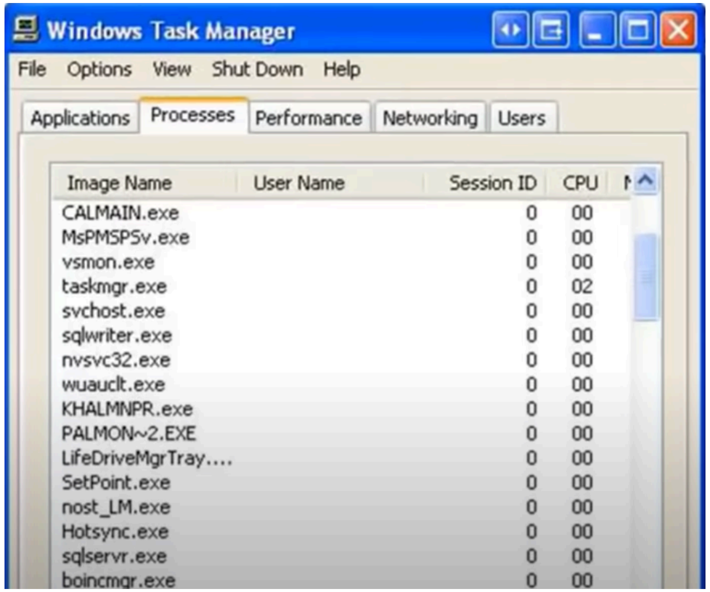
LS-10. Process Management.

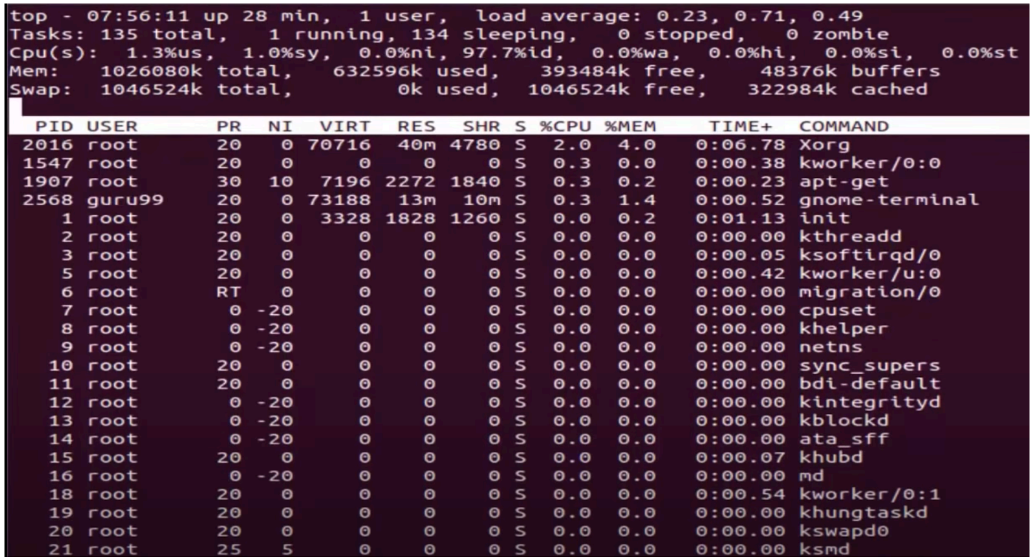


Process Management and Multithreading

Agenda

- 1. Operating System Processes
- 2. CPU Scheduling
- 3. Process Operation on Linux
- 4. Manage a Linux running process
- 5. Introduction to Threads
- 6. Process Management Implementations
- 7. Process Synchronization
- 8. Deadlocks





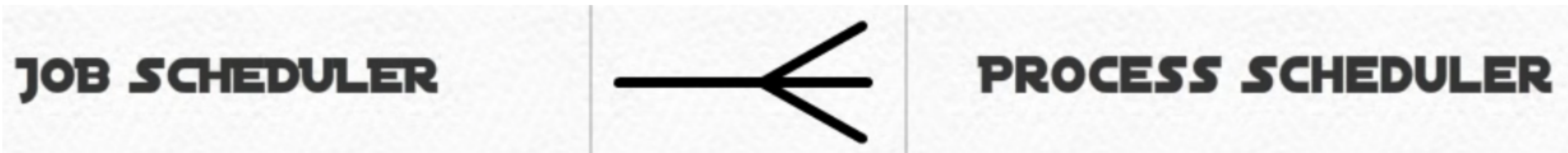
1. Operating System Processes

- What is a Process?

A program in the execution is called a Process. Process is not the same as program. A process is more than a program code. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.

- Process memory is divided into 4 sections for efficient working:

1. The text section is made up of the compiled program code, read in from non-volatile storage when the program is launched.
2. The data section is made up the global and static variables, allocated and initialized prior to executing the main.
3. The heap is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
4. The stack is used for local variables. Space on the stack is reserved for local variables when they are declared.



1.1. Operations on Process

- A system that manages process must be able to perform certain operations and with the process. These include 7 state process model.

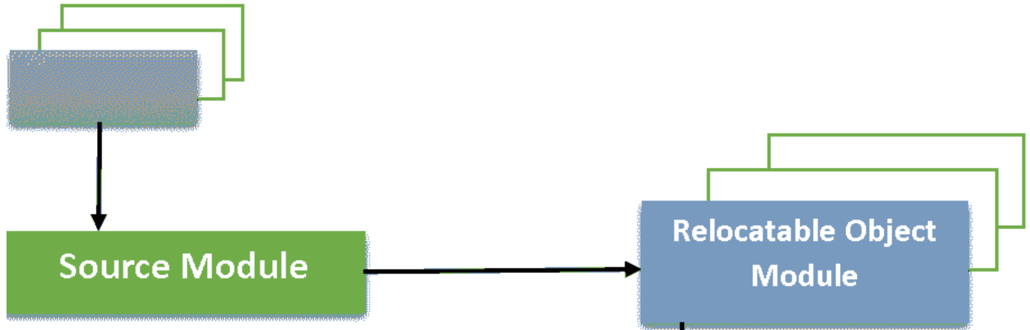
1. Create a process
2. Destroy (Terminate) a process
3. Resume a process (Restart the process)
4. Change the priority of a process
5. Block of process
6. Wake up a process
7. Enable a process to communicate with other processes

1.1. Operations on Process

Process creation in OS

When a user initiates to execute a program, the operating system creates a process to represent the execution of this program.

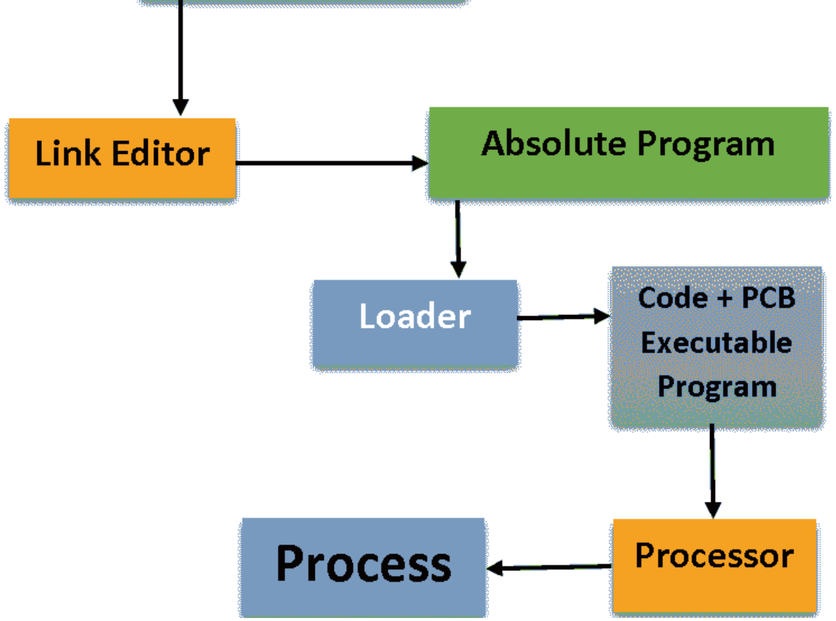
The creation of executable programs include many steps.



The relocatable object modules are converted into absolute programs by the linker.

Absolute programs are converted into executable programs by the loader, and then the processor executes these programs.

The program in execution is called a process. The process consists of the program in memory, plus the PCB(Process Control Block) structure.



1.1. Operations on Process

- Process Termination

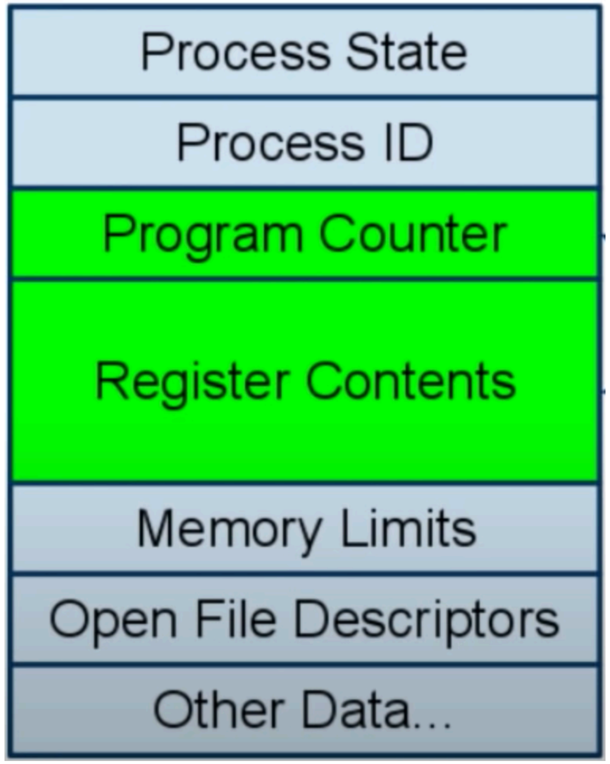
The process terminates from the running state includes so many causes

1. Process execution is finished
2. Time slot expired
3. Memory boundary violation
4. Input/Output failure
5. Parent termination
6. Parent request
7. Invalid instruction

1.2. Process Control Block

There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following:

- 1. Process State - It can be running, waiting etc.
- 2. Process ID and parent process ID.
- 3. CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.
- 4. CPU Scheduling information - Such as priority information and pointers to scheduling queues.
- 5. Memory Management information - Eg. page tables or segment tables.
- 6. Accounting information - user and kernel CPU time consumed, account numbers, limits, etc.
- 7. I/O Status information - Devices allocated, open file tables, etc.

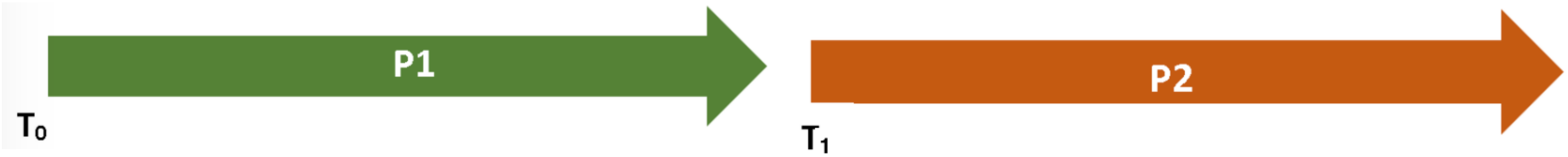


Program Counter and Register Contents - used for saving process CPU state when switching processes.

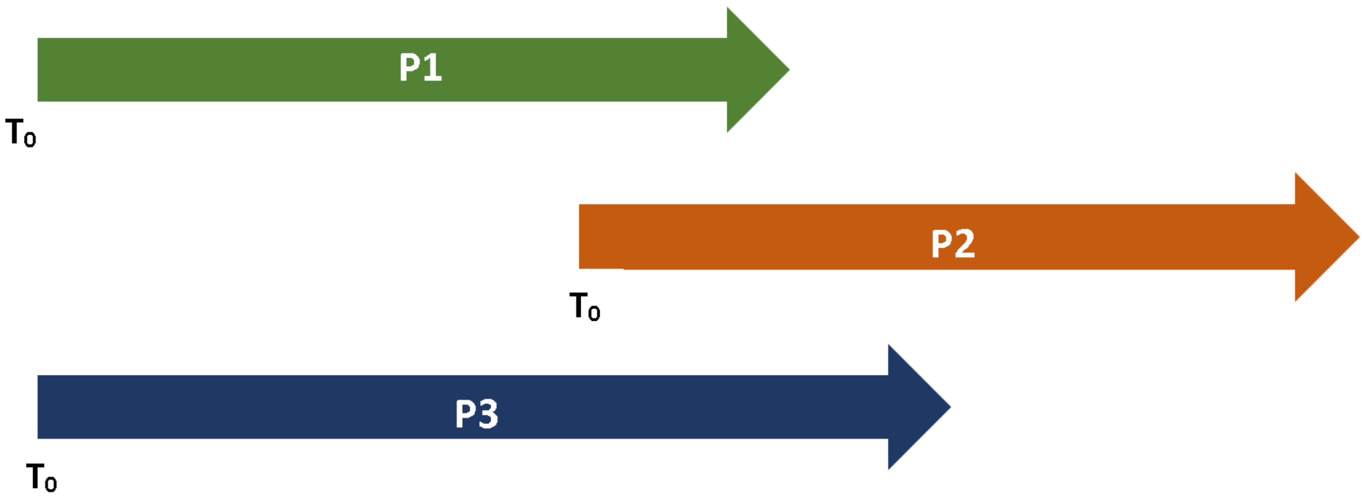
1.3. Switching Processes

1.3.1. Concurrent Process

- Two processes are 'serial' if the execution of one must be completed before the execution of other starts.



- If the two or more processes said to be concurrent, they are not serial, and their execution can overlap in time.



1.3. Switching Processes

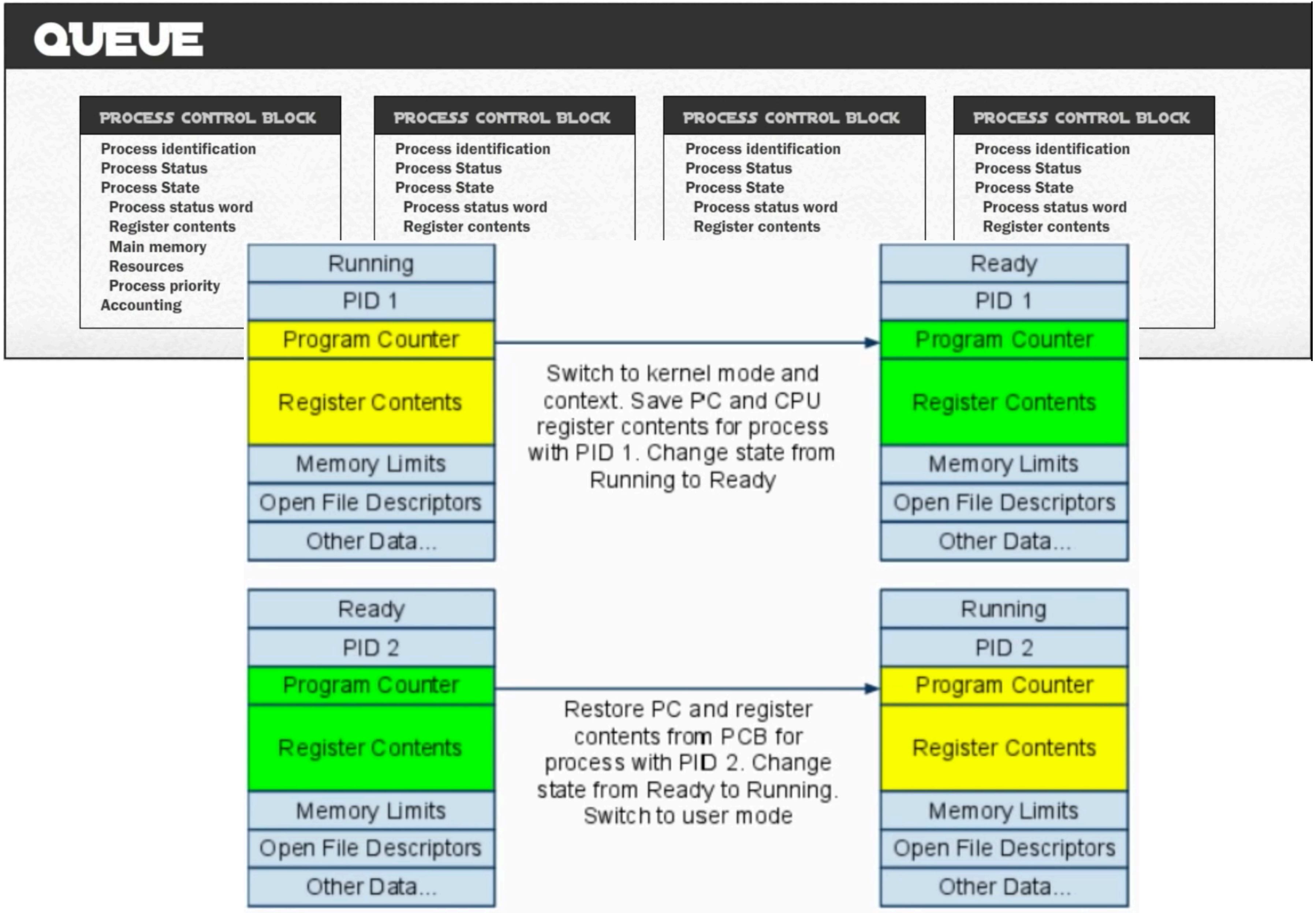
1.3.2. Process Context

- Minimal set of state information that must be stored to allow a process to be stopped and re-started later
- Information stored in the CPU
  - Contents of registers
  - Program Counter (PC) (aka Instruction Pointer)
- Information stored in RAM (SWAP)

1.3.3. Context and Process Switches

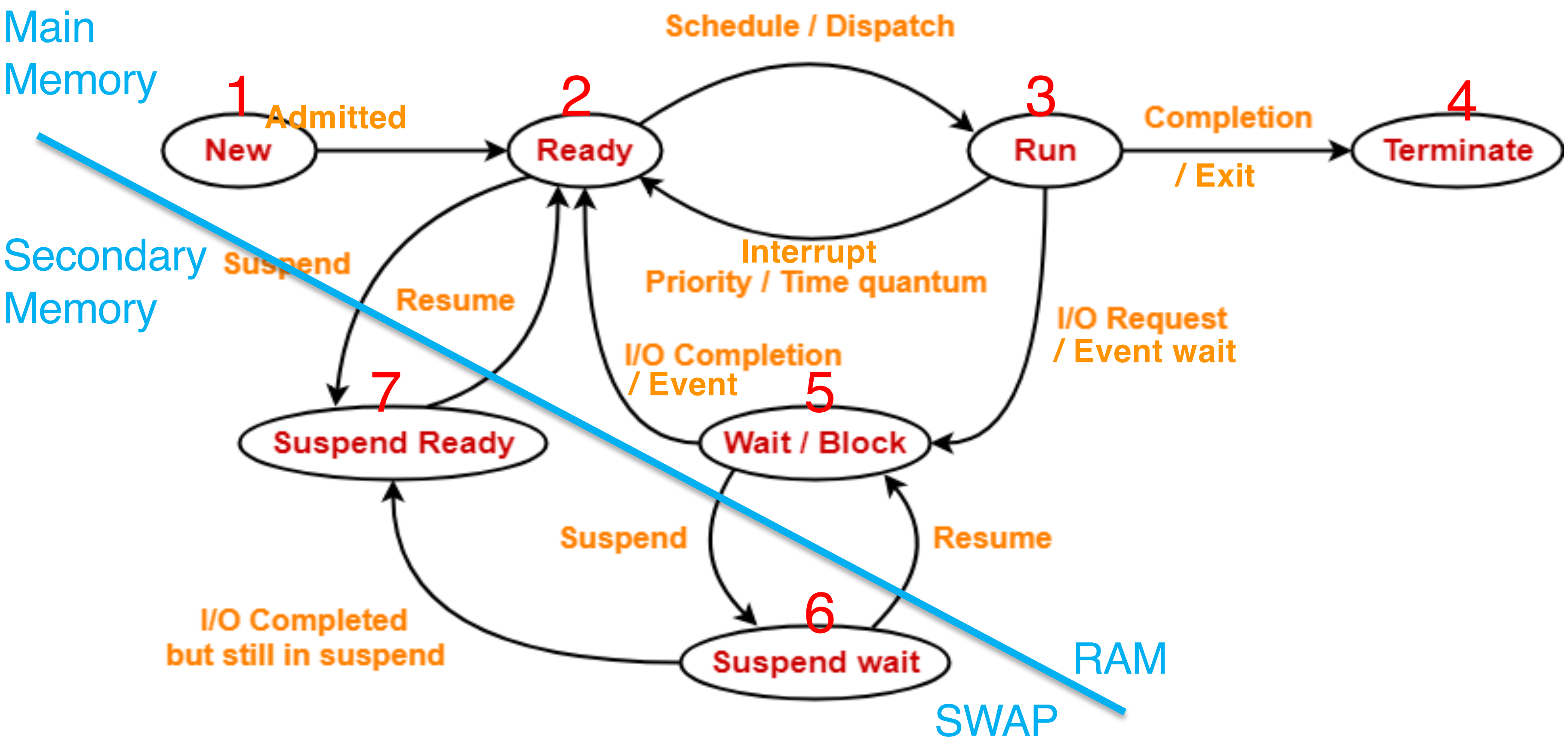
- Context Switch
  - System switches from running a process to running kernel code
  - Computationally expensive operation
- Process Switch
  - Operating system switches from one process to another
  - Requires saving the state of one process, then restoring the state of another process
  - Two context switches required – into kernel and out of kernel

1.3. Switching Processes



1.4. Process States

Processes can be any of the following states



Process State Diagram

1.4. Process States

- **1. New State** - A process is said to be in new state (1) when a program present in the secondary memory is initiated for execution.
- **2. Ready State** - A process moves from new state (1) to queue (очередь) of ready state (2) after it is loaded into the main memory and is ready for execution. In ready state, the process waits for its execution by the processor. In multiprogramming environment, many processes may be present in the ready state.
- **3. Run State** - A process moves from ready state (2) to run state (3) after it is assigned the CPU for execution.
- **4. Terminate State** - A process moves from run state (3) to terminate state (4) after its execution is completed. After entering the terminate state, context of the process (process descriptor) is deleted by the operating system.
- **5. Block Or Wait State** - A process moves from run state (3) to block or wait state (5) if it requires an I/O operation or some blocked resource during its execution. After the I/O operation gets completed or resource becomes available, the process moves to the ready state (2).
- **6. Suspend Wait State** - A process moves from wait state (5) to suspend wait state (6) if a process with higher priority has to be executed but the main memory is full.

  Moving a process with lower priority from wait state to suspend wait state clear a room for higher priority process in the ready state.

  After the resource becomes available, the process is moved to the suspend ready state (7). After main memory becomes available, the process is moved to the ready state.
- **7. Suspend Ready State** - A process moves from ready state (2) to suspend ready state (7) if a process with higher priority has to be executed but the **main memory is full** (Memory Swapping).

  Moving a process with lower priority from ready state to suspend ready state clear a room for higher priority process in the ready state.

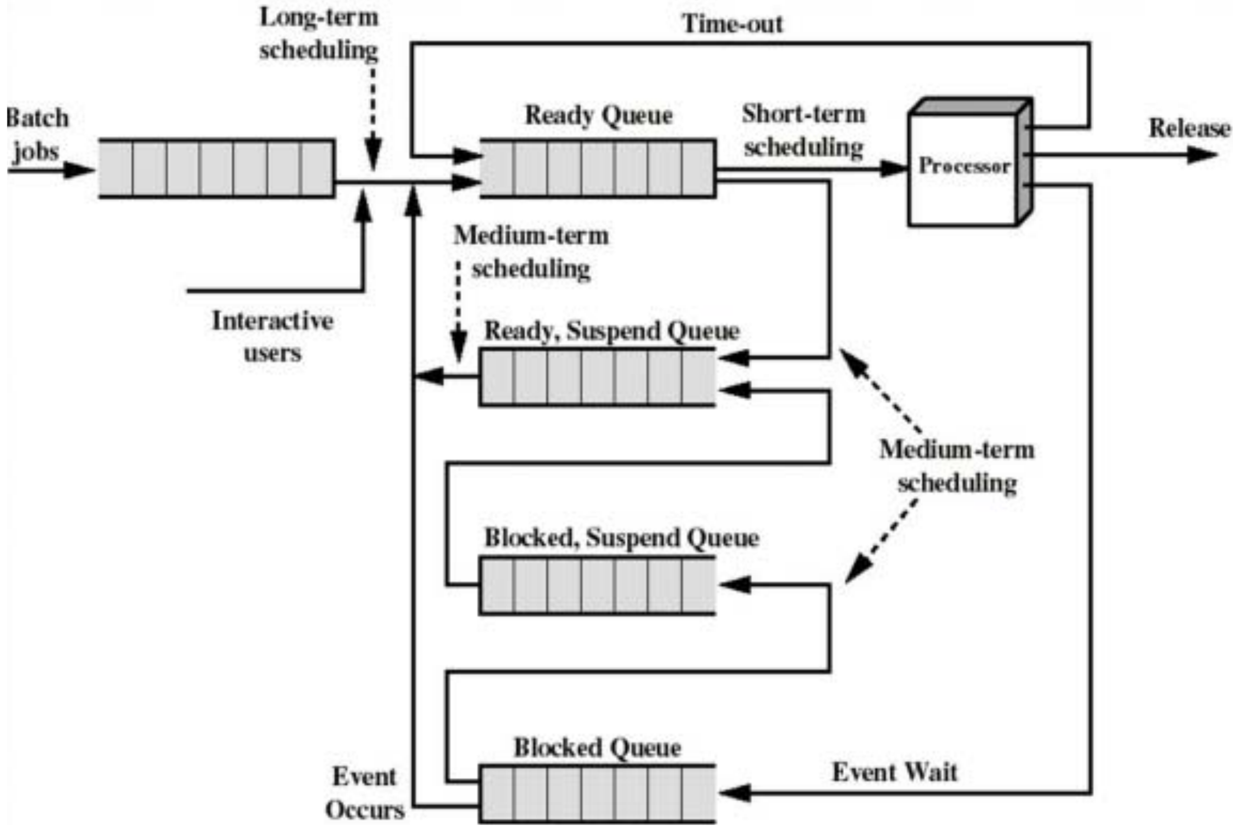
  The process remains in the suspend ready state until the main memory becomes available. When main memory becomes available, the process is brought back to the ready state (2).

1.5. Processes Queues

1.5.1. Scheduling Queues

- All processes when enters into the system are stored in the **batch queue**.
- Processes in the Ready state are placed in the **ready queue**.
- Processes waiting for a device or event to become available are placed in **suspended queues**. There are unique queues for each I/O device or each event available.

1.5.2. System Queues and Long, Medium, Short Terms Schedulers.

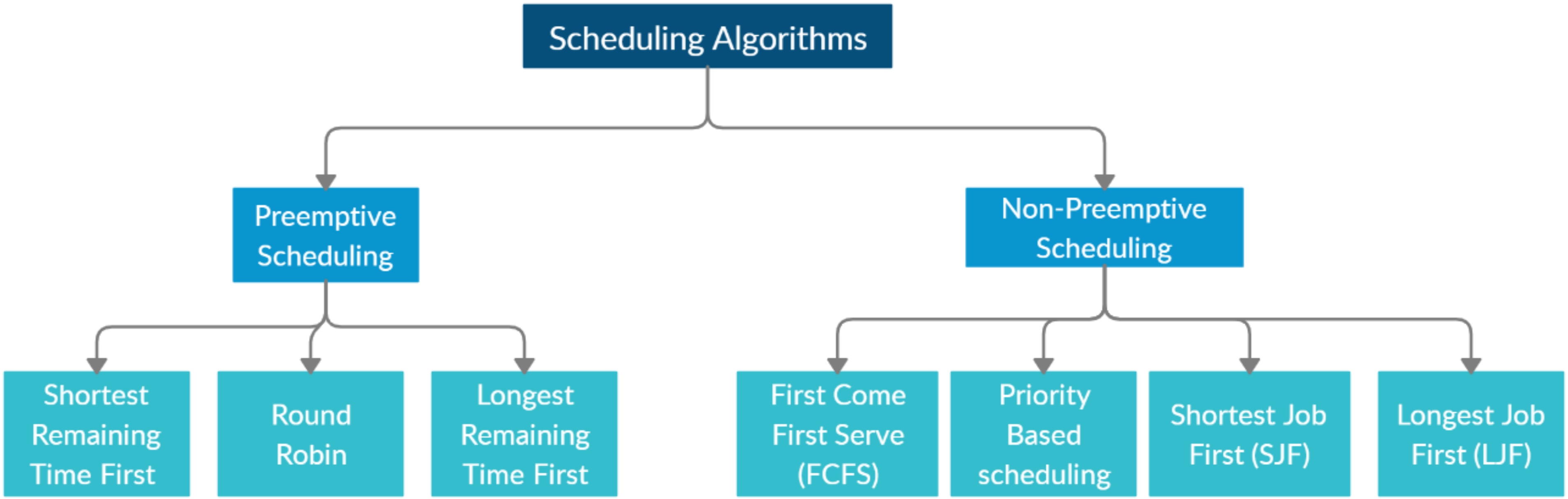


1.6. Process Scheduling

- The act of determining which process in the ready state should be moved to the running state is known as Process Scheduling.

1.6.1. Schedulers fell into one of the two general categories:

- **Preemptive scheduling.** When the operating system decides to favor another process, preempting the currently executing process.
- **Non preemptive scheduling.** When the currently executing process gives up the CPU voluntarily.



1.6. Process Scheduling

1.6.2. There are three types of schedulers available:

- **Long Term Scheduler** (Batch Job Scheduler):
  - Long term scheduler runs less frequently.
  - Long intervals (seconds to minutes)
  - Selects processes to bring into ready list
  - Average rate of process creation is equal to the average departure rate of processes from the execution memory.
- **Mid-Term Scheduler** (Suspend/Swap Scheduler):
  - Swaps inactive processes out to disk
  - Restores swapped processes from disk on demand
  - Selects processes to bring into ready list
- **Short Term Scheduler** (CPU Scheduler):
  - Runs very frequently
  - Short intervals (milliseconds)
  - Selects processes from ready list to run on CPU cores
  - The primary goal of this scheduler is increase process execution rate.

1.6. Process Scheduling

1.6.3. Tasks of the process scheduling for different system types.

- 1. For all system types
  - Fairness - every process gets a fair share of CPU time
  - Balance - keeping all parts of the system busy (for example: keeping the processor and I/O devices busy)
- 2. Batch processing systems
  - Throughput - the number of tasks per hour
  - Turnaround time - minimizing the time spent waiting for service and processing tasks.
- 3. Interactive systems
  - Response time - quick response to requests
  - Proportionality - fulfilling the user's expectations (for example: the user is not ready for a long system load)
- 4. Real time systems
  - Deadline Completion - Prevent Loss of Data Value
  - Predictability - preventing quality degradation in multimedia systems (for example: loss of audio quality should be less than video)

2. CPU Scheduling

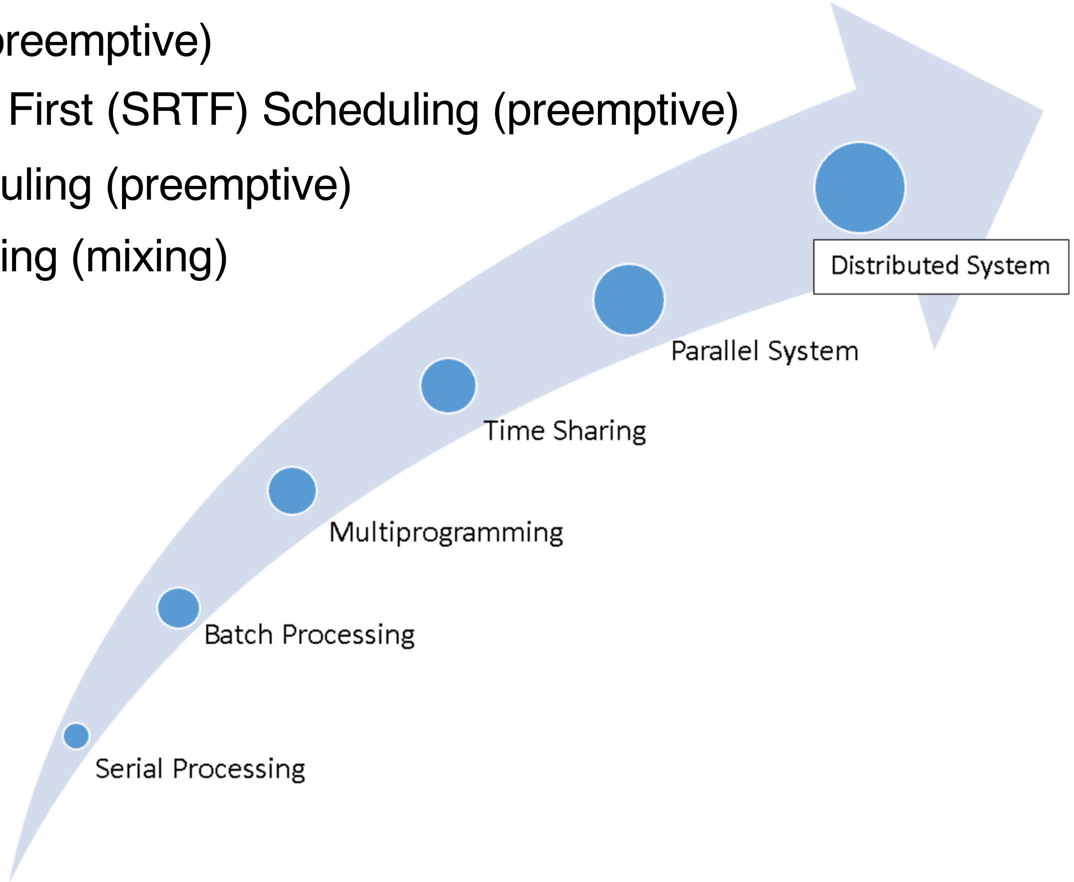
- The goal of CPU scheduling is to make the system efficient, fast and fair.
- CPU Scheduling Criteria to check when considering the "best" algorithm:
  - **1. CPU utilization.** To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time (Ideally 100%). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
  - **2. Throughput (пропускная способность).** It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.
  - **3. Turn Around Time** - total time taken to finish process execution.
  - **4. Waiting Time** - the sum of time process waits in ready or In/Out waiting queues to acquire get control on the CPU.
  - **5. Load average.** It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.
  - **6. Response Time** - process time taken when process gets CPU for the first time.
  - **7. Arrival Time** - when process enters Ready queue form Job Queue.
  - **8. Burst Time** - CPU time required by the process to complete execution.

What is Burst, Arrival, Response, Waiting, Turnaround times and Throughput? Read on After Academy:

https://afteracademy.com/blog/what-is-burst-arrival-exit-response-waiting-turnaround-time-and-throughput

2. CPU Scheduling

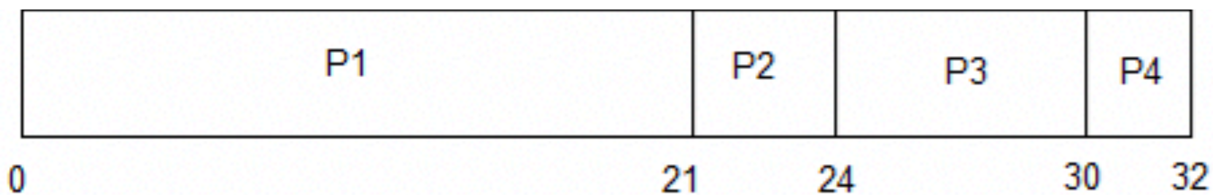
- Major CPU Scheduling Algorithms:
  - 1. First Come First Serve (FCFS) Scheduling (non preemptive)
  - 2. Shortest Job First (SJF) Scheduling (non preemptive)
  - 3. Priority Scheduling (non preemptive)
  - 4. Shortest Remaining Time First (SRTF) Scheduling (preemptive)
  - 5. Round Robin (RR) Scheduling (preemptive)
  - 6. Multilevel Queue Scheduling (mixing)



2.1. First Come First Serve (FCFS) Scheduling

- Jobs are executed on FCFS basis.
- Easy to understand and implement.
- Poor performance because Twait is high.
- Burst Time refers to the time required in milliseconds by a process for its Execution (CPU time of a process).
- Processes table and Gantt chart →

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



Twaiting=Tstarting - Tarrival (P1=0, P2=21-0, P3=24-0, P4=30-0)
TwaitingAvg=TwaitingAllProcess/Nprocess=(0+21+24+30)/4= 18.75 ms

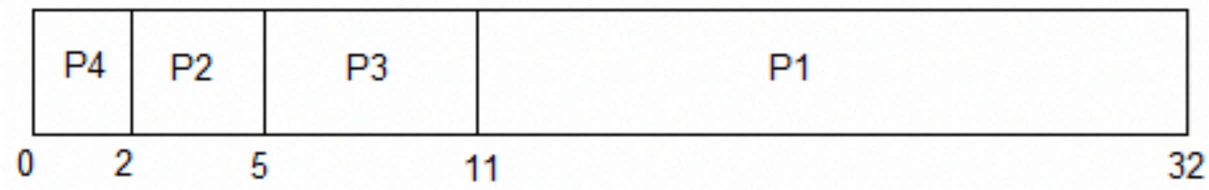
Tturnaround=TwaitReadyQueue+Texecution+TwaitInOutQueue
TturnaroundTotal=(0+21+0)+(21+3+0)+(24+6+0)+(30+2+0)= 107 ms
TturnaroudAvg=TturnaroundTotal/Nprocess=107/4= 26.75 ms

Throughput=(21+3+6+2)/4= 8 ms (one process executes every 8 ms)

2.2. Shortest Job First (SJF) Scheduling

- In SJF shortest process is executed first.
- Best algorithm to minimize waiting time.
- Processes of the same length run in FCFS mode.
- Difficult to implement since the system does not know the Burst time of the process.
- Processes table and Gantt chart →

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



Twaiting=Tstarting - Tarrival (P4=0, P2=2-0, P3=5-0, P1=11-0)
TwaitingAvg=TwaitingAllProcess/Nprocess=(0+2+5+11)/4= 4.5 ms

Tturnaround=TwaitReadyQueue+Texecution+TwaitInOutQueue
TturnaroundTotal=(0+2+0)+(2+3+0)+(5+6+0)+(11+21+0)= 50 ms
TturnaroudAvg=TturnaroundTotal/Nprocess=50/4= 12.5 ms

Throughput=(2+3+6+21)/4= 8 ms (one process executes every 8 ms)

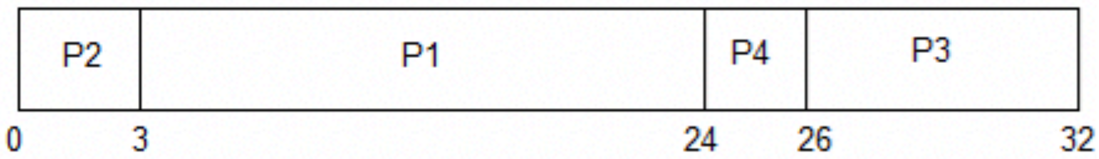
2.3. Priority Scheduling (non-preemptive)

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS mode.
- Priority can be decided based on:
  - memory requirements,
  - time requirements,
  - any other resource requirement.
- Disadvantage

The major problem with priority scheduling is starvation (голодание процесса), because low priority jobs are waiting for the CPU for a long time.

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

- Processes table and Gantt chart →



TwaitingAvg=(0+3+24+26)/4= 13.25 ms

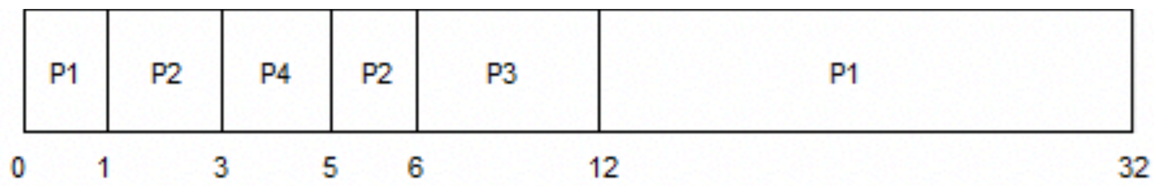
Tturnaround=TwaitReadyQueue+Texecution+TwaitInOutQueue
TturnaroundTotal=(0+2+0)+(3+21+0)+(24+2+0)+(26+6+0)= 84 ms
TturnaroudAvg=TturnaroundTotal/Nprocess=84/4= 21 ms

Throughput=(3+21+2+6)/4= 8 ms (one process executes every 8 ms)

2.4. Shortest Remaining Time First (SRTF)Scheduling

- In SRTF, jobs are put into ready queue as they arrive.
- If there is a process with a short burst time, the existing process is preempted (вытесняется).
- Processes of the same length run in FCFS mode.
- TwaitAvg for SRTF is less than both, SJF and FCFS.
- Processes table & Gantt chart →

PROCESS	BURST TIME	ARRIVAL TIME
P1	21	0
P2	3	1
P3	6	2
P4	2	3



Twaiting=Tstarting - Tarrival (P1=12-1, P2=5-3, P3=6-2, P4=3-3)
TwaitingAvg=TwaitingAllProcess/Nprocess=(11+2+4+0)/4= 4.25 ms

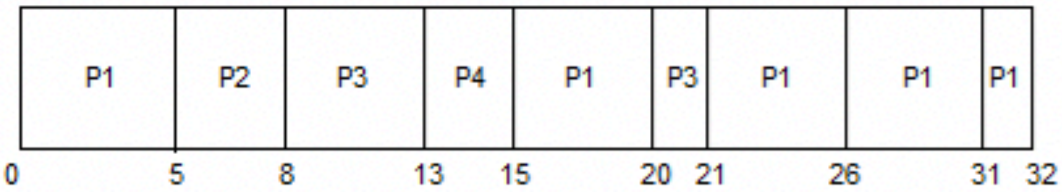
Tturnaround=TwaitReadyQueue+Texecution+TwaitInOutQueue
TturnaroundTotal=(11+21+0)+(2+3+0)+(4+6+0)+(0+2+0)= 49 ms
TturnaroudAvg=TturnaroundTotal/Nprocess=49/4= 12.25 ms

Throughput=(1+2+2+1+6+20)/4= 8 ms (one process executes every 8 ms)

2.5. Round Robin (RR) Scheduling (preemptive)

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preempted and other process executes for given time period.
- Processes of the same length run in FCFS mode.
- Context switching is used to save states of preempted processes.
- Processes table and Gantt chart →
- Example for Time Quantum = 5 ms

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



Twaiting=SUMquant(Tstarting - Tarrival)
P1=0-0+15-5+21-20+26-26+31-31
P2=5-0
P3=8-0+20-13
P4=13-0
TwaitingAvg=(11+5+20+13)/4= 11 ms

Tturnaround=TwaitReadyQueue+Texecution+TwaitInOutQueue
TturnaroundTotal=(16+21+0)+(5+3+0)+(15+6+0)+(13+2+0)= 81 ms
TturnaroudAvg=TturnaroundTotal/Nprocess=81/4= 20.25 ms

Throughput=(5+3+5+2+5+1+5+5+1)/4= 8 ms (one process executes every 8 ms)

2.6. Multilevel & Multilevel Feedback Queue Scheduling

Multilevel Queue Scheduling

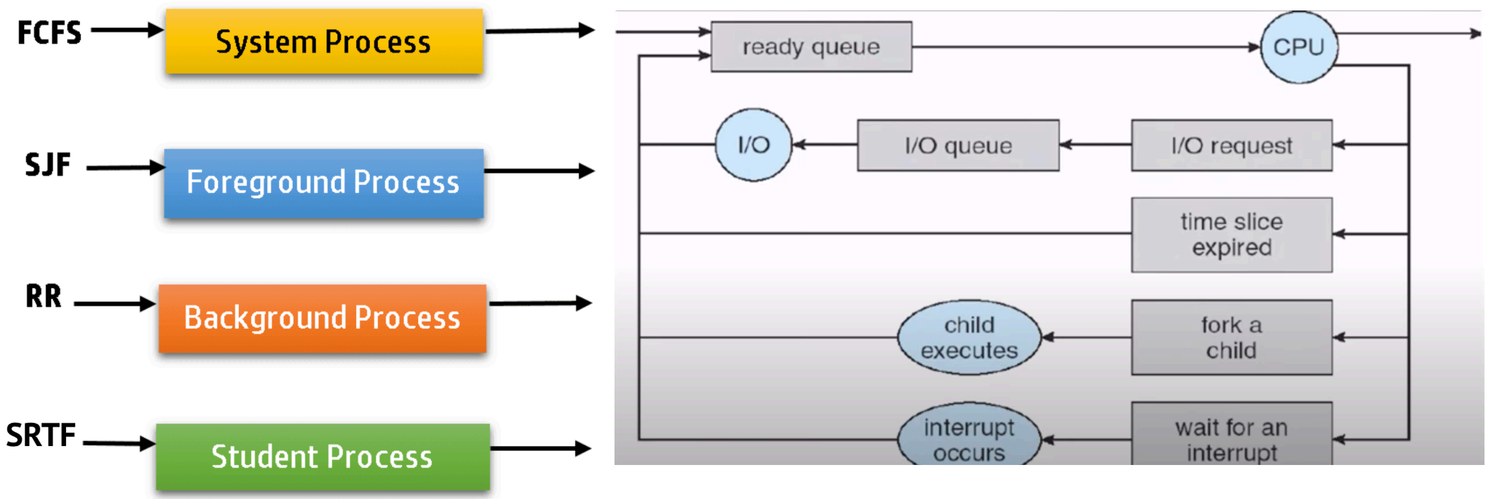
- Multilevel Queue Scheduling combine a advantages of many algorithms.
- Multiple Ready queues are maintained for processes.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue (multi-priority).

Multilevel Feedback Queue Scheduling

- The algorithm who maximizes the CPU utilization and throughput, and minimizes the turnaround time, waiting time and response time, are the best of all.
- Scheduling algorithm can allows a process to move between the queues, if process wait long time
- Variable time quantum's can used for every process.

Multiple Queues usage and queuing diagram of Process Scheduling

Highest Priority



Lowest Priority

2.7. CPU Scheduling Conclusion

- We should choose our algorithm such that no processes starve for the resource and minimize the average waiting time, average response time and average turnaround time.

- FCFS may cause long waiting time.
- SJF and SRTF may cause process starvation (голодание).
- Round Robin scheduling algorithm will behave as FCFS if time quantum is large.
- Multilevel Queue Scheduling combine a advantages of many algorithms.
- Multilevel Feedback Queue Scheduling algorithms is a best scheduling algorithm.

2.8. Scheduling Problems Examples

Tasks 2.7.1. FCFS

- TwaitAvg= 25 ms
- TthurnaroundAvg= 38.4 ms
- Throughput= 11.6 ms

Process	Burst Time(ms)
P_1	5
P_2	24
P_3	16
P_4	10
P_5	3

P_1	P_2	P_3	P_4	P_5

0 5 29 45 55 58

Task 2.7.2. SJF

- TwaitAvg= 12.6 ms
- TthurnaroundAvg= 24.2 ms
- Throughput= 11.6 ms

Process	Burst Time(ms)
P_1	5
P_2	24
P_3	16
P_4	10
P_5	3

P_5	P_1	P_4	P_3	P_2

0 3 8 18 34 58

Task 2.7.3. SRTF

- TwaitAvg= 3.2 ms
- TthurnaroundAvg= 7.2 ms
- Throughput= 4 ms

Process	Burst Time(CPU)	Arrival Time(ms)
P_1	3	0
P_2	6	2
P_3	4	4
P_4	5	6
P_5	2	8

P_1	P_2	P_3	P_5	P_2	P_4

0 3 4 8 10 15 20

Task 2.7.4. Priority

- TwaitAvg= 10 ms
- TthurnaroundAvg= 15.2 ms
- Throughput= 5.2 ms

Process	CPU Burst Time	Priority
P_1	6	2
P_2	12	4
P_3	1	5
P_4	3	1
P_5	4	3

Gantt Chart

P_4	P_1	P_5	P_2	P_3

0 3 9 13 25 26

Task 2.7.5. RR

- TwaitAvg= 15 ms
- TthurnaroundAvg= 29.66 ms
- Throughput= 14.66 ms

Process	CPU Burst Time
P_1	30
P_2	6
P_3	8

Gantt Chart

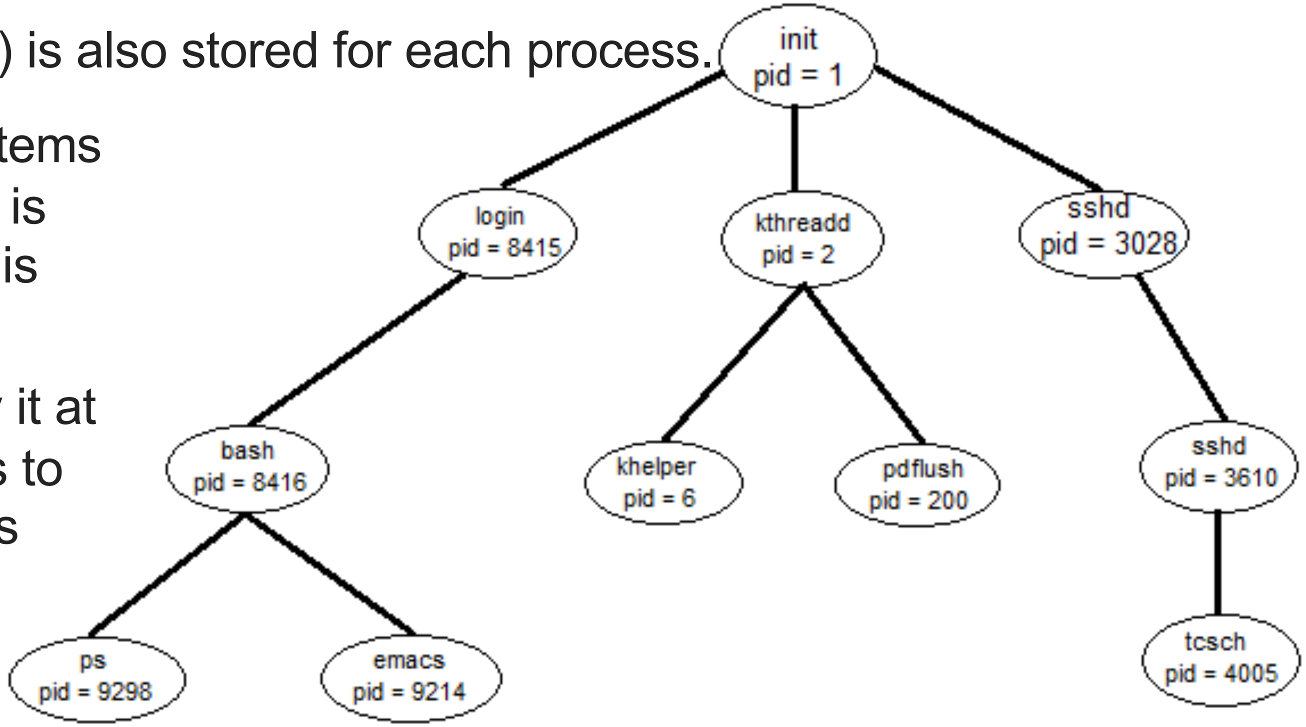
P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_1	P_1	P_1

0 5 10 15 20 21 24 29 34 39 44

3. Process Operations on Linux

3.1. Process Creation.

- Through appropriate system calls, such as fork or spawn, processes may create other processes. The process which creates other process, is termed the parent of the other process, while the created sub-process is termed its child.
- Each process is given an integer identifier, termed as process identifier, or PID.
- The parent PID (PPID) is also stored for each process.
- On a typical UNIX systems the process scheduler is termed as sched, and is given PID 0.
- The first thing done by it at system start-up time is to launch init, which gives that process PID 1.
- Process is created via fork() or exec().



A Tree of processes on a typical Linux system

3. Process Operations on Linux

- Further Init launches all the system daemons and user logins, and becomes the ultimate parent of all other processes.
- A child process may receive some amount of shared resources with its parent depending on system implementation. To prevent runaway children from consuming all of a certain system resource, child processes may or may not be limited to a subset of the resources originally allocated to the parent.
- There are two options for the parent process after creating the child:
  - Wait for the child process to terminate before proceeding. Parent process makes a wait() system call, for either a specific child process or for any particular child process, which causes the parent process to block until the wait() returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
  - Run concurrently with the child, continuing to process without waiting. When a UNIX shell runs a process as a background task, this is the operation seen. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation.

3. Process Operations on Linux

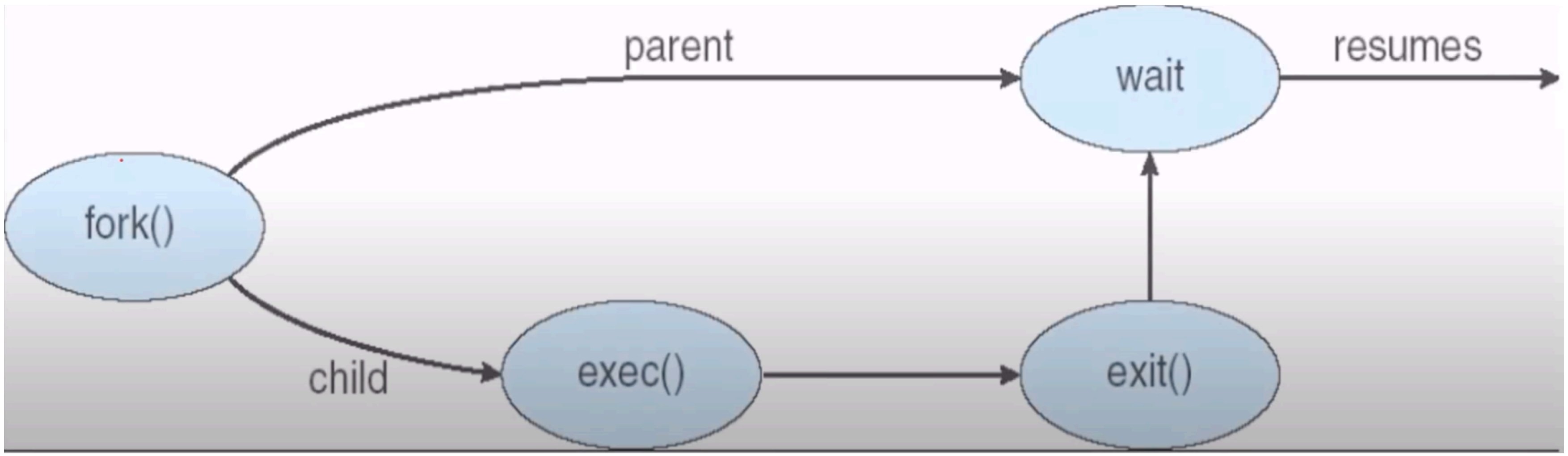
3.2. Process Termination

- By making the exit(system call), typically returning an int, processes may request their own termination. This int is passed along to the parent if it is doing a wait(), and is typically zero on successful completion and some non-zero code in the event of any problem.
- Processes may also be terminated by the system for a variety of reasons, including:
  - The inability of the system to deliver the necessary system resources.
  - In response to a KILL command or other unhandled process interrupts.
  - A parent may kill its children if the task assigned to them is no longer needed i.e. if the need of having a child terminates.
  - If the parent exits, the system may or may not allow the child to continue without a parent (In UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them.)
- When a process ends, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process already became an orphan.
- The processes which are trying to terminate but cannot do so because their parent is not waiting for them are termed **zombies**. These are eventually inherited by init as orphans and killed off.

3. Process Operations on Linux

3.3. Process related system calls (in Unix)

- fork() creates a new child process
  - All processes are created by forking from a parent.
  - The init process is ancestor of all processes.
  - After fork, parent and child are running same code
- exec() used after fork() to replace the process memory space with a new program code
- exit() terminates a process (parent clean child resources)
- wait() causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments



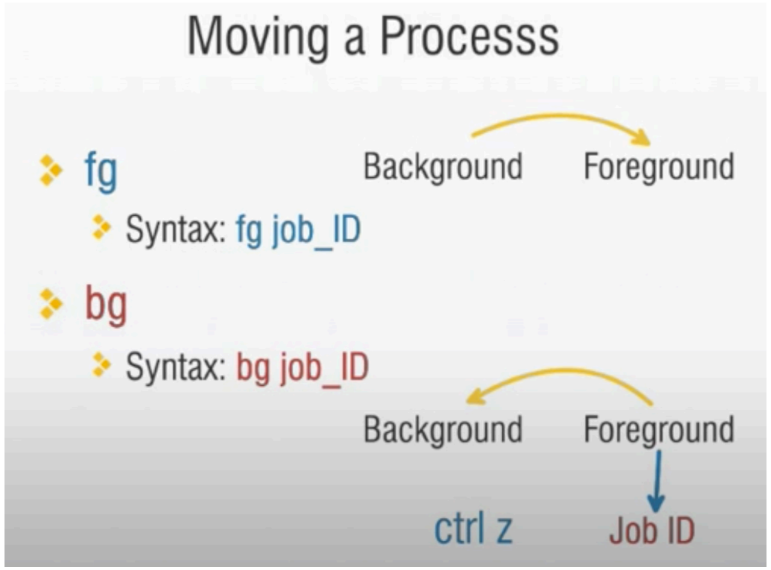
4. Manage a Linux running process

4.1. Start, stop Jobs and Processes

```
$ cat file               #foreground process=job
$ cat file | wc -l       #foreground job=two processes
$ cat file | wc -l &     #background job=two processes
[1] 2543                 #job ID and last process ID
$ ping abc.lv            #foreground process=job
$ Ctrl+c                 #SIGTERM to current process

$ yes > /dev/null &
[2] 2556
$ jobs -l                #jobs listing
[1]+  Done      ls -l | wc -l
[2]+ Running    yes > /dev/null &
```

4.2. fg, bg



To bring a background process to the foreground

```
$ fg  OR fg %2  #move to foreground
```

To move a foreground process in the background:

1. Stop the process by typing Ctrl+z.
2. Move the stopped process to the background by typing bg.

```
$ Ctrl+z        #SIGHUP to current process
$ bg            #move to background
```

4. Manage a Linux running process

4.3. nohup

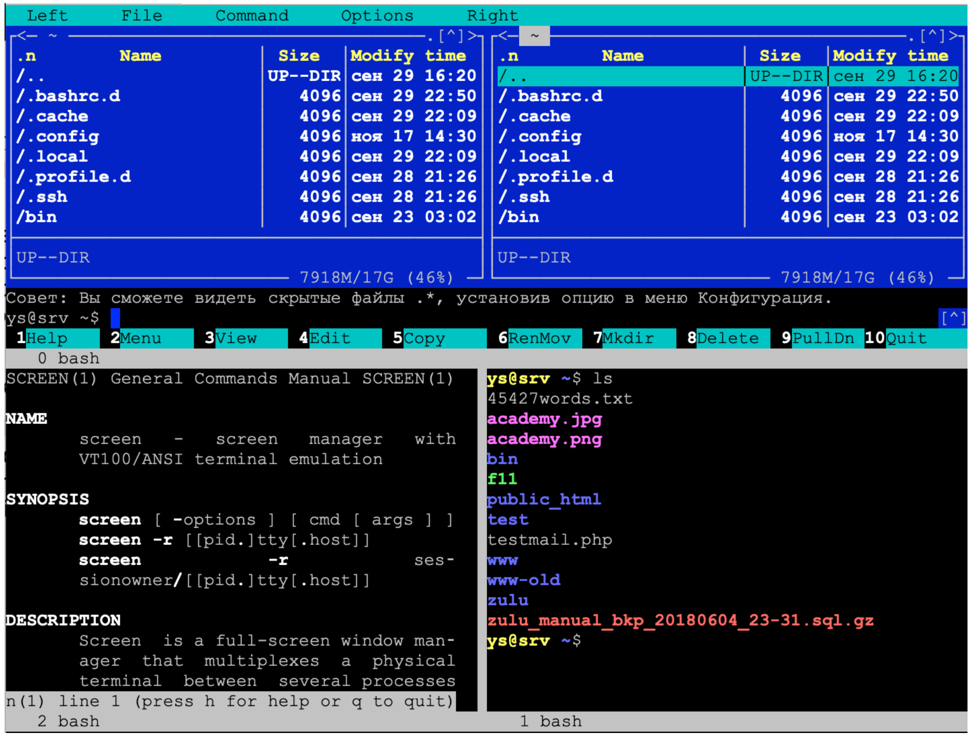
A process may not continue to run when you log out or close your terminal. This special case can be avoided by preceding the command you want to run with the nohup command. Also, appending an ampersand (&) will send the process to the background and allow you to continue using the terminal. Nohup does is return the running process's PID. Result write or to redirect file > res-file, or to ./nohup.out, or to ~/nohup.out

```
$ nohup   ping abc.lv > res-file &      #keeping a process running
2654
```

4.4. screen (modern nohup analog)

Screen is a terminal multiplexer program that allows you to start a screen session and open any number of windows (virtual terminals) inside that session. Processes running in Screen will continue to run when their window is not visible even if you get disconnected.

```
$ screen            #Starting Unnamed Session
$ Ctrl+a ?          #Screen help
$ Ctrl+a d          #Detach from screen session
$ screen -S name    #Starting Named Session
$ screen -ls        #Session listing
$ screen -r PID     #Reattach to a Screen PID
$ Ctrl+a c          #Create new terminal instance
$ Ctrl+a n          #Next Screen
$ Ctrl+a p          #Previous Screen
$ Ctrl+a S          #Split current region horiz
$ Ctrl+a |          #Split current region vertic
$ Ctrl+a tab        #Switch input to next region
```



4. Manage a Linux running process

4.5. ps

The default output of ps is a simple list of the processes running in your current terminal. As you can see below, the first column contains the PID.

```
$ ps
PID         TTY         TIME         CMD
23058       pts/0       00:00:00  bash
23069       pts/0       00:00:00  ps
```

ps to show me every running process (**-e**) and a full listing (**-f**)

```
$ ps -ef  #every running process (-e), full listing (-f)
UID          PID  PPID  C STIME TTY        TIME CMD
root           1     0  0 Nov16 ?      00:00:19 /sbin/init
root           2     0  0 Nov16 ?      00:00:00 [kthreadd]
root           3     2  0 Nov16 ?      00:00:00 [rcu_gp]
…
root      23039   576  0 14:18 ?          00:00:00 sshd: ys [priv]
root      23045     2  0 14:18 ?          00:00:00 [kworker/0:1]
ys        23057 23039  0 14:18 ?          00:00:00 sshd: ys@pts/0
ys        23058 23057  0 14:18 pts/0      00:00:00 -bash
root      23170   576  0 14:22 ?          00:00:00 sshd: unknown [priv]
…
sshd      23193 23191  0 14:22 ?          00:00:00 sshd: unknown [net]
ys        23209 23058  0 14:23 pts/0      00:00:00 ps -ef
```

4.6. pstree

```
ys@srv ~$ pstree -u ys
screen───bash

sshd───bash─┬─mc───bash───pstree
            └─yes
```

4.7. top - viewing details of running processes and quickly identifying problem (memory and other).

```
$ top                                                                         0. modificators -H, -c, -u
top - 14:30:47 up 1 day, 11:42,  1 user,  load average: 0,08, 0,08, 0,03     1.
Tasks: 136 total,   1 running, 135 sleeping,   0 stopped,   0 zombie          2.
%Cpu(s):  0,3 us,  0,0 sy,  0,0 ni, 99,7 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st  3.
MiB Mem :    482,4 total,     11,3 free,    215,4 used,    255,7 buff/cache   4.
MiB Swap:    512,0 total,    501,4 free,     10,6 used.    246,3 avail Mem    5.
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND  6.
  808 mysql     20   0 1274152  65160   5176 S   0,3  13,2   0:34.55 mysqld
23057 ys        20   0   16192   5024   3972 S   0,3   1,0   0:00.13 sshd
    1 root      20   0  104048   7228   4928 S   0,0   1,5   0:19.19 systemd
    2 root      20   0       0      0      0 S   0,0   0,0   0:00.00 kthreadd
    3 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 rcu_par_gp
    6 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 kworker/0:0H-kblockd
    8 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 mm_percpu_wq
    9 root      20   0       0      0      0 S   0,0   0,0   0:07.48 ksoftirqd/0
   10 root      20   0       0      0      0 I   0,0   0,0   0:12.37 rcu_sched
```

Understanding top's interface:

0. Sort-View - press KEY: M- by memory%, P by cpu%, N by PID, T by Time; H by threads statistic (default tasks statistic), R - ascending order, v -forest

1. SysStatistic: Sys time, Uptime, User sessions, Load Average of CPU over 1, 5, 15 min – number of running processes, example, 0.4 = 40%/coreNr.

2. Process Statistic: Running Processes & Processes State

3. CPU usage in %: user, system, manual changed nice, idle, In/Out wait, hardware $ system interrupt event wait, VM steal time on Virtual Environment

4. Memory usage – total, free, for processes used RAM, for disk buff/cache used RAM

5. SWAP usage – total, free, for processes used SWAP and for processes available RAM (without swap, but include cache usage)

6. Task Area: PID, EUID, PRiority, Nice, VIRT - all memory, RES – RAM, SHR – share memory with other processes, State, %CPU, %MEM, live TIME

6a. Processes States:
(R) Runnable: A process in this state is either executing on the CPU, or it is present on the Run Queue, ready to be executed.
(S) Interruptible Sleep: Processes in this state are waiting for an event to complete (Event Queue).
(D) Uninterruptible Sleep: In this case, a process is waiting for an I/O operation to complete (In/Out Queue).
(T) Stopped: These processes have been stopped by a job control signal (such as by pressing Ctrl+Z).
(Z) Zombie: Terminated processes whose data structures are still around and parent is not around are called zombies.

4. Manage a Linux running process

4.8. kill

Kill is used to send a signal to a process. The most commonly used signal is "terminate" (SIGTERM) or "kill" (SIGKILL). However, there are many more. Below are some examples. The full list can be shown with **kill -L**.

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
```

The default signal is 15, which is SIGTERM

```
$ kill 20896
```

Notice signal number nine is SIGKILL. Usually, we issue a command such as

```
$ kill -9 20896
$ kill -15 20226 1823 26785
$ kill -s KILL 3245
```

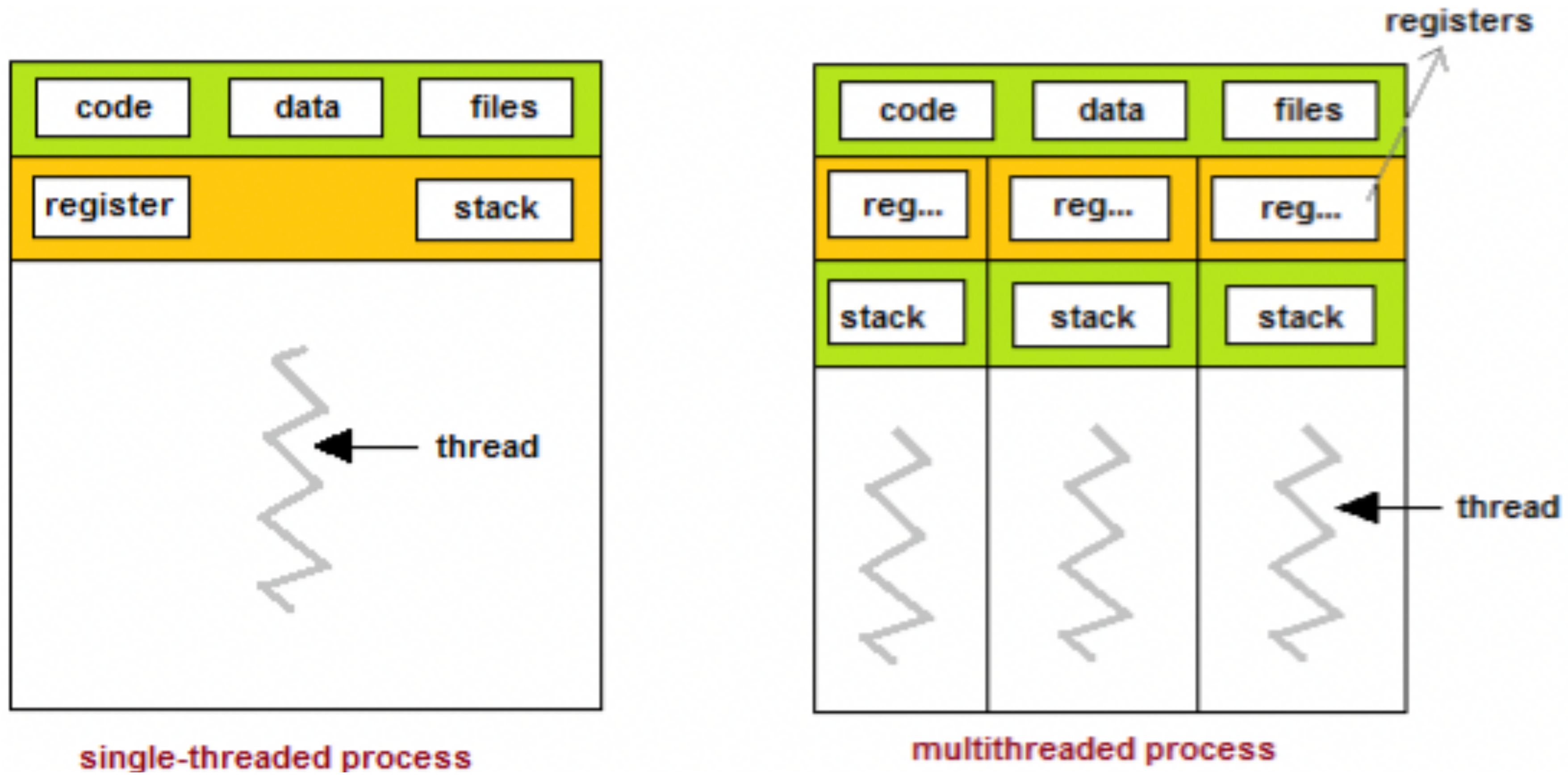
Keep in mind that many applications have their own method for stopping.

4.9. nise, renice

```
$ nice -11 yes>/dev/null& #set the priority of a command yes
$ renice 5 20901          #changing priority of the proc. 20901
$ top -u student          #check the nice value of a process
# nice --10 command       #set the negative priority for a cmnd.
# renice -n 15 -p 235     #changing priority of the proc. 235
```

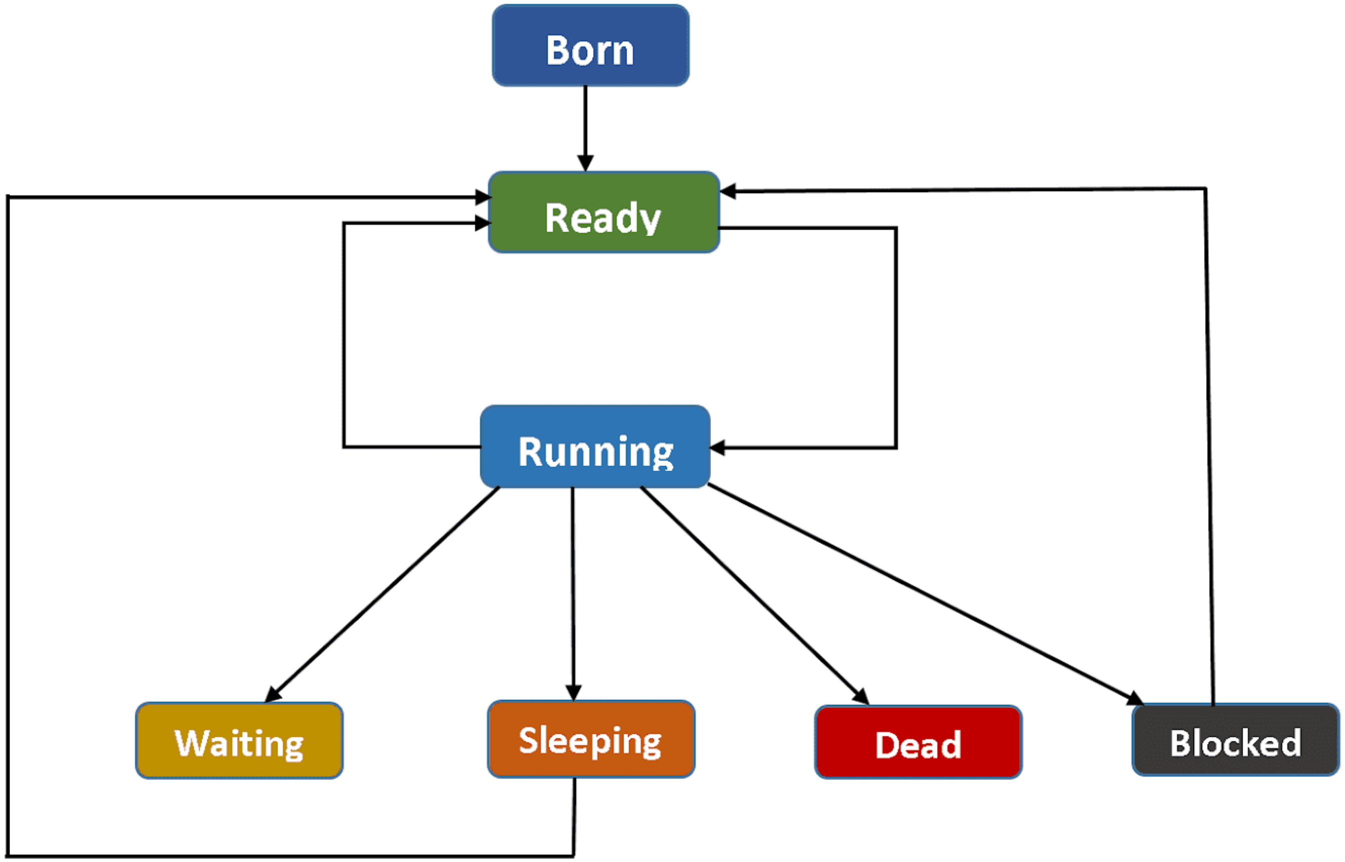
5. Introduction to Threads

- Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Lightweight processes.
- Threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.



5. Introduction to Threads

5.1. Thread Life Cycle in OS

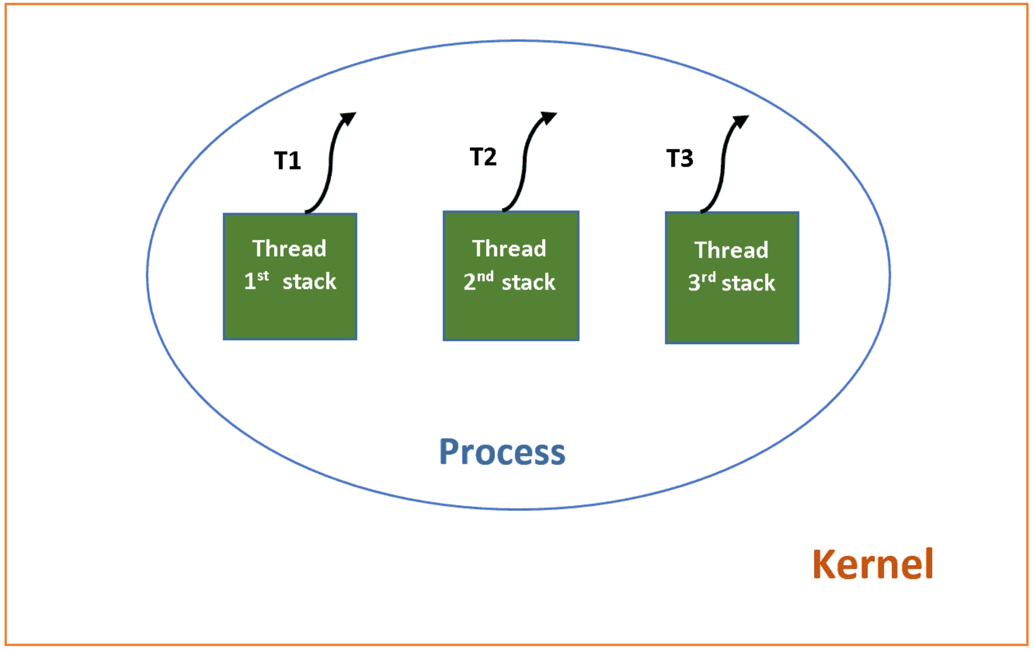


5. Introduction to Threads

5.2. Multithreading in OS

Multithreading in an operating system divided into four categories:

1. **One-To-One Model.** One Process, One Thread: In this traditional approach, the process maintains only one thread. For example, the MS-DOS operating system supports this approach.
2. **Many-To-One Model.** Multi Processes, One Thread: Operating system supports multiple user processes but only support one thread process. For example UNIX.
3. **One-To-Many Model.** One Process, Multi Threads: In this approach, a process divided into the number of threads. For example, Java Runtime Environment.
4. **Many-To-Many Model.** Multi Processes, Multi Threads: In this approach, a process divided into the number of threads. For example Window 2000, Solaris, LINUX.



5. Introduction to Threads

5.3. Types of Thread

User threads, are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

Kernel threads are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

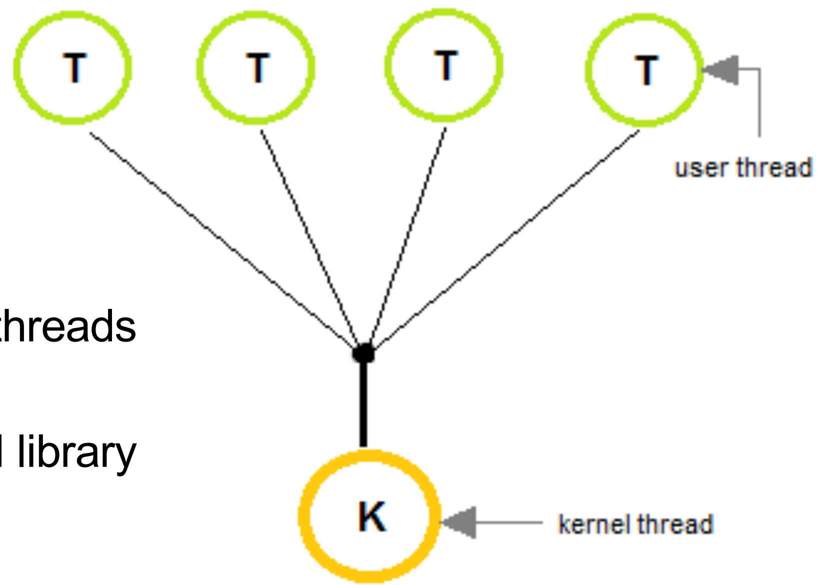
5.4. Multithreading Models

The user threads must be mapped to kernel threads, by following strategies:

- **Many-To-One Model**
- **One-To-One Model**
- **Many-To-Many Model**

5.4.1. Many-To-One Model

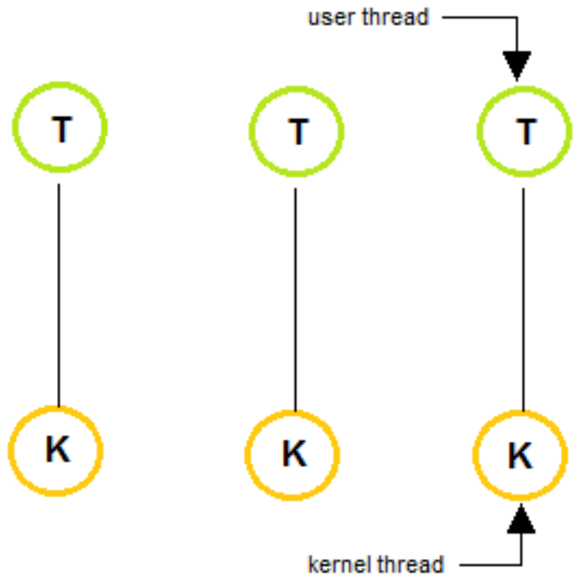
- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is efficient in nature.



5. Introduction to Threads

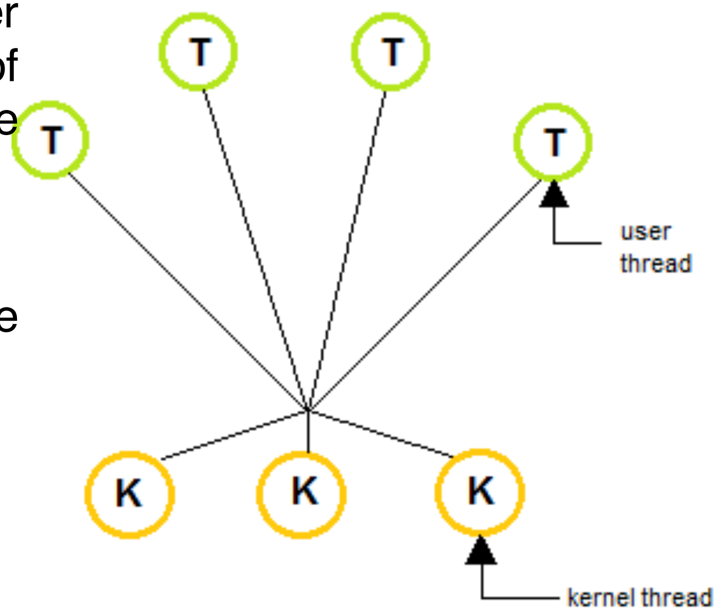
5.4.2. One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



5.4.3. Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.



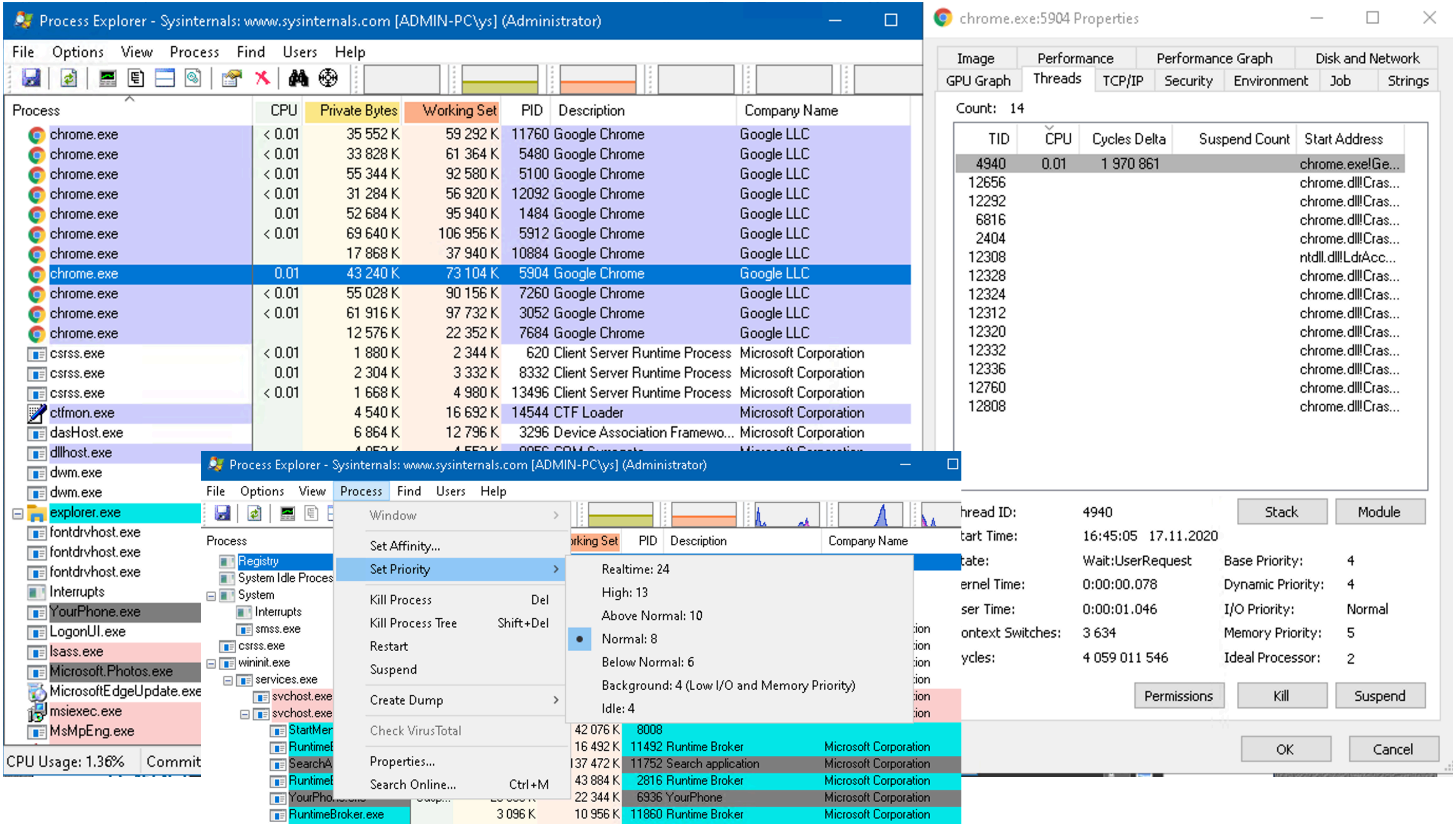
5. Introduction to Threads

5.5. Difference Between Process and Thread in OS

- A process cannot share the same memory space whereas; threads can share memory and files.
- It takes more time to create a process whereas; it takes less time to create a thread.
- The process takes more time to complete the execution and termination whereas; thread takes less time to terminate.
- Process execution is slow, but threads execute very fast.
- Context switching time between two processes is much whereas; context switching time between two threads is less as compared to the process.
- Implementing the communication between two processes is more difficult, but communication between the two threads is easy to implement because threads share the memory.
- System calls are required to communicate with each process, but in the case of a thread, system calls not necessary.
- The loosely coupled process, but tightly coupled threads.
- The process requires more resources to execute whereas; the thread requires fewer resources to execute. Therefore, the thread is called a lightweight process.
- A process is not suitable for parallel activity-based whereas threads are suitable for the parallel activity.

5. Introduction to Threads

5.6. Processes and Threads in Windows (Process Explorer)



6. Process Management Implementations

■ 6.1. OS NetWare 4.x

non-preemptive algorithm

Чтобы не занимать процессор слишком долго, поток в NetWare сам отдает управление планировщику ОС, используя следующие системные вызовы:

1) Thread Switch – поток считает себя готовым к выполнению немедленно, но по длительным меркам отдает управление ОС

2) ThreadSwitchWithDelay – аналогично предыдущей, но считает что будет готов через несколько переключений с потока на поток.

3) Delay – функция аналогична предыдущей, но задержка дается в миллисекундах.

4) ThreadSwitchLowPriority – считает себя готовым к выполнению немедленно, но также и низкоприоритетным.

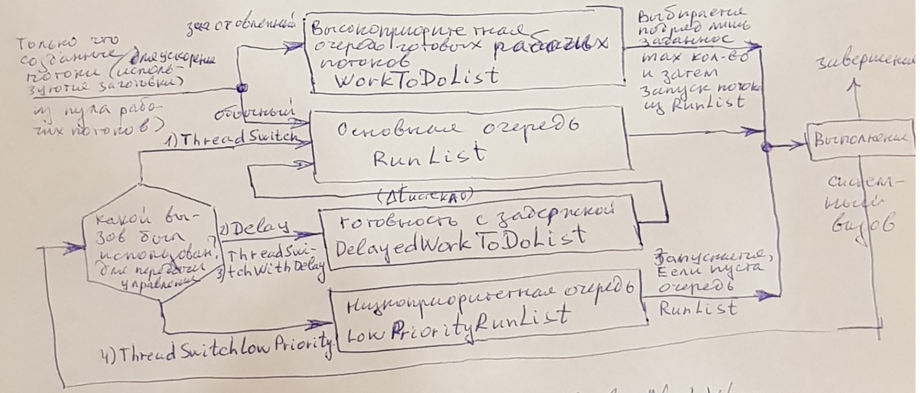


Схема планирования потоков в NetWare

Общий итог

Из-за небольших накладных расходов ОС на диспетчеризацию потоков за счёт простых алгоритмов планирования и иерархии контекстов, данная ОС потенциально очень производительна.

Но для достижения высокой производительности к разработчикам приложений для этой ОС предъявляются высокие требования.

6. Process Management Implementations

- 6.2. OS-9

mix algorithm

тстов в OS-9. В этой ОС каждый процесс имеет статически определенный приоритет и возраст. Возраст (age) – количество просмотров очереди с того момента, когда этот процесс в последний раз получал управление. Обе эти характеристики представлены 16 разрядными беззнаковыми числами.

Процесс может повысить или понизить свой статич. приоритет, исполнив соответствующие системные вызовы, но система по собственной инициативе никогда не меняет его (приоритет процесса).

- Управление получает процесс с наибольшей суммой статического приоритета и динамически изменяющегося возраста.

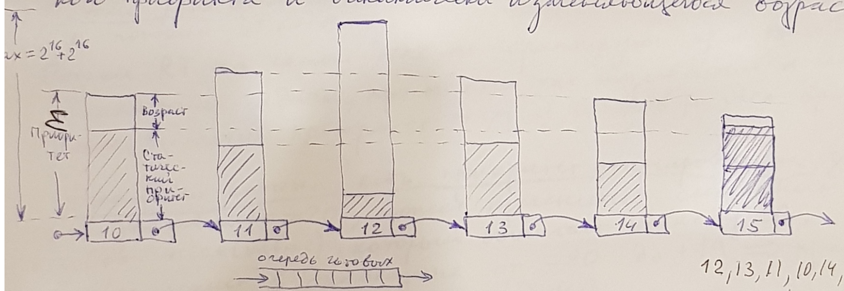


рис. Динамическое изменение приоритета в OS/9

На рисунке показан пример: Будет выбран 12 процесс, хотя его статический приоритет самый маленький.

- Если у процессов одинаковый суммарный приоритет (см. процессы 10 и 14), то выбирается с более высоким статическим (т.е. 10).

- Если у процессов совпадают суммарные и статические приоритеты (см. процессы 11 и 13), то выбирается стоящий ближе к началу очереди (справа), т.е. 13. (очередь представлена связн. списком)
  т.е. раньше поступивший в систему

Этот алгоритм гарантирует, что любой низкоприоритетный процесс рано или поздно получит управление.

Если необходимо, чтобы процесс получал управление чаще, то нужно просто повысить его статический приоритет.

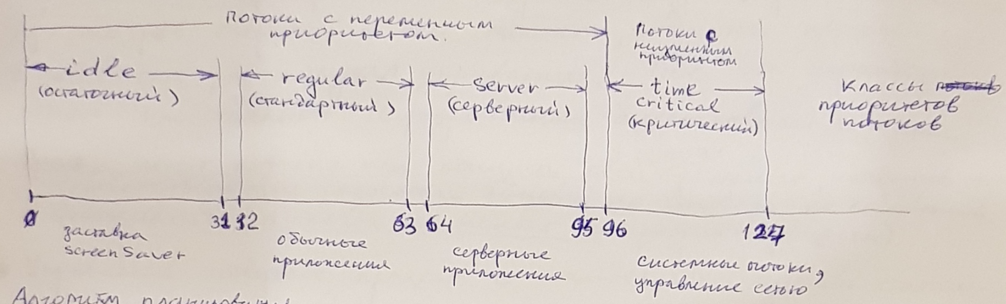
Возможна более тонкая регулировка, если ~~имеется~~ системный вызов, запрещающий исполнение процесса ниже заданного приорит., или ограничить max возраст (не стареть больше 100

6. Process Management Implementations

- 6.3. OS/2

mix algorithm

Планирование основано на использовании квантования и абсолютных динамических приоритетов. OS/2 поддерживает понятие процесса и потока.



Алгоритм планирования

- Выбирается поток из очереди наивысшего класса (всегда)
- Внутри класса выбирается поток с наивысшим приоритетом.
- Потоки с одинаковым приоритетом обслуживаются циклически.

"Справедливость" и отсутствие "голодания" ~~про~~ низкоприоритетных потоков реализуется планировщиком ОС.

- Если поток ожидает процессор дольше чем величина, заданная в системной переменной MAXWAIT, то его приоритет автоматически увеличивается ОС, но не более **96** (нижняя граница time critical).
- Если поток ушел в режим ожидания операции ввода-вывода, то после ее завершения он получит наивысший приоритет в своем классе.
- Приоритет потока автоматически повышается, когда он поступает на выполнение в систему.

ОС динамически устанавливает величину кванта, отводимого потоку для выполнения. Параметры настройки системы либо её загрузка могут регулировать величину кванта в пределах 32 мс – 65536 мс.

- Если поток был прерван (вытеснен) до истечения кванта, то следующий выделенный квант будет увеличен на 32 мс (один период таймера) и так до max величины, заданной при настройке ОС. MAXквант

Благодаря такому алгоритму планирования в OS/2 ни один поток не будет "забыт" системой и получит достаточное процессорное время.

6. Process Management Implementations

6.4. Linux

mix algorithm

Планирование в Linux отличается от алгоритмов, используемых в ОС UNIX. Потоки в Linux реализованы в ядре, поэтому планирование основано на потоках, а не на процессах.

ОС Linux различает 3 класса потоков:

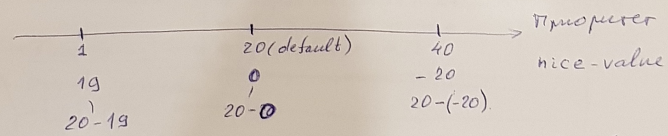
1. Потоки реального времени, обслуживаемые по FIFO
2. Потоки реального времени, обслуживаемые по циклу FIFO с прерыванием по таймеру.
3. Потоки разделения времени.

1. Потоки реал. врем. FIFO имеют наивысший приоритет и не могут прерываться другими потоками, за исключением такого же потока, перешедшего в состояние готовности.

2. Потоки реал. врем. циклические могут прерываться таймером (отработав квант времени) и потоком RT FIFO.

Потоки RT на самом деле не гарантируют предельный срок выполнения задачи, просто они так называются и имеют более высокий приоритет чем у потоков раздел. времени.

У каждого потока есть приоритет планирования, значение по умолчанию = 20, может изменяться системным вызовом nice(value), который вычитает value из 20. value лежит в диапазоне от -20 до +19. ⇒ приоритеты лежат в промежутке от 1 до 40.



Цель алгоритма планирования – обеспечить соответствие качества обслуживания потока его приоритету:

- ↑ приоритет – ↓ отклик
- ↑ приоритет – ↑ процессорного времени.

Для обеспечения второго пункта с каждым потоком связан ~~квант~~ динамический квант времени (количество тиков таймера, в течение которых процесс может выполняться) По умолчанию системные часы тикают с частотой 100 Гц ⇒ тик = 10 мс.

Общая схема планирования в Linux показана на рисунке. Планировщик Linux
- поддерживает очереди готовых процессов
- расчитывает для них добродетель
- выбирает поток
- выделяет потокам квант
- пересчитывает квант

6. Process Management Implementations

- 6.5. Windows

mix algorithm

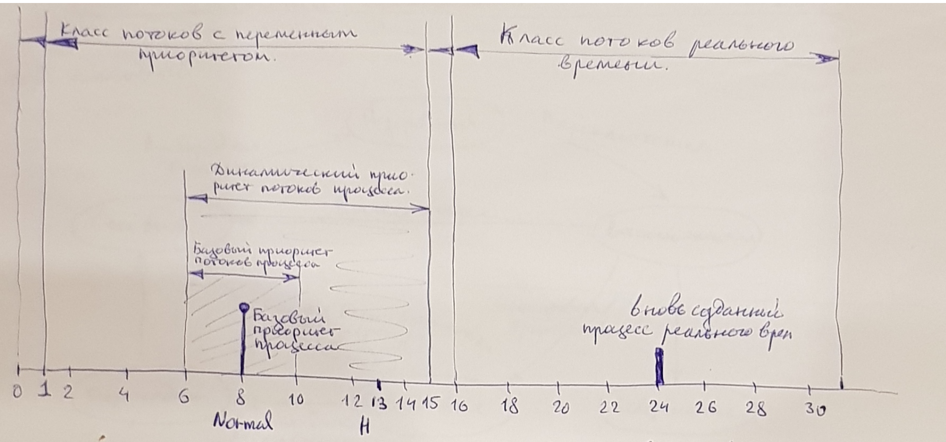


Схема назначения приоритетов в Windows NT/2000

Базовый приоритет каждого потока определяется по классу приоритета процесса и по уровню приоритета потока внутри класса своего процесса.

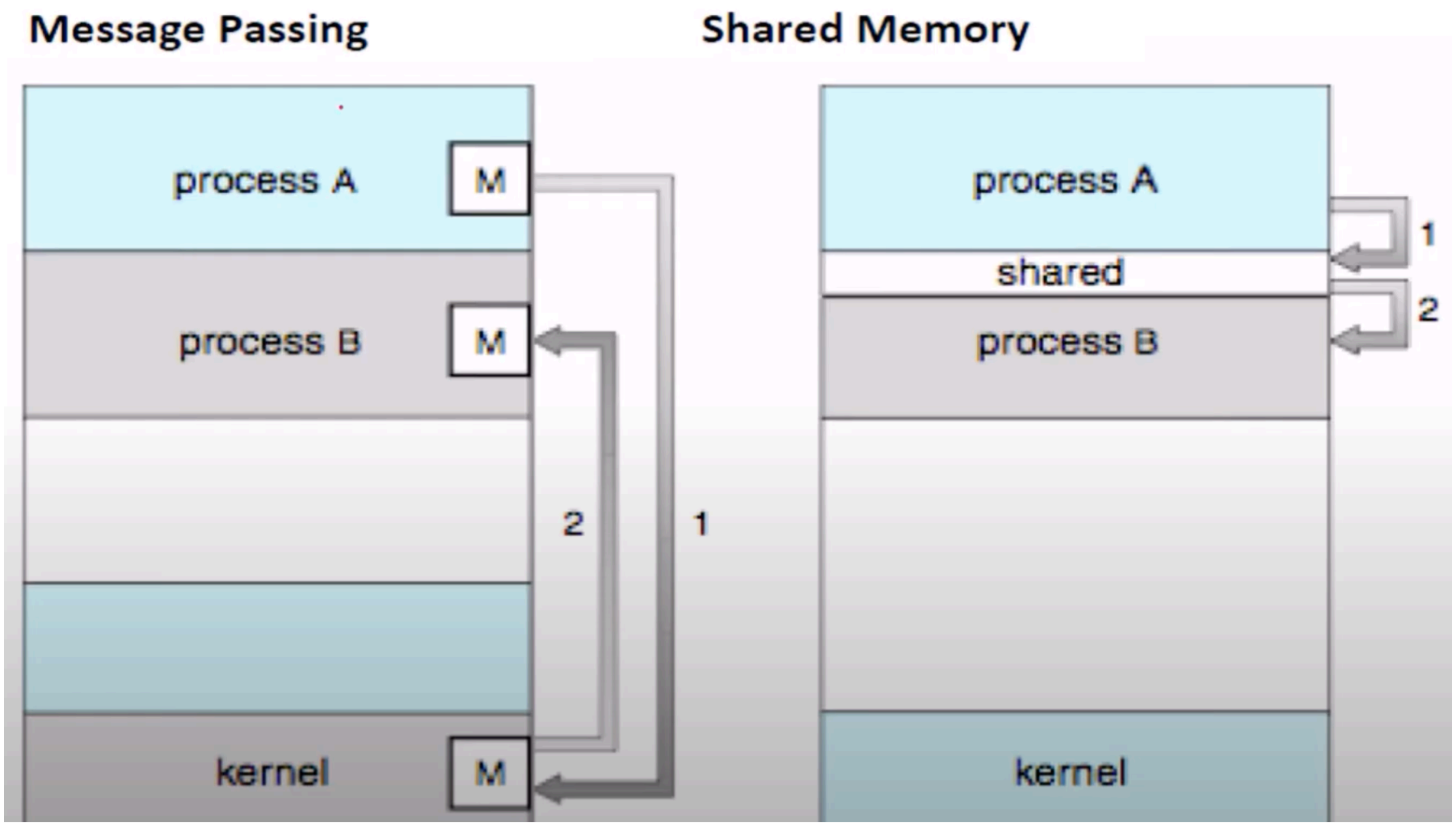
- При создании процесса он по умолчанию получает тот или иной класс: реального времени (системные) (16-31) или переменного приоритета (пользовательские) (1–15). 0 зарезервирован системой.
- Базовый приоритет процесса присваивается разработчиком приложения и может быть повышен или понижен ОС, а значит изменятся и приоритеты потоков, связанных с процессом.
- Первоначально процессам могут присваиваться след. базовые уровни.

Realtime	= 24	(16-31)	базовые уровни и диапазоны приоритетов.
High	= 13	(11-15)	
Normal	= 8	(6 – 10)	
Low (Idle)	= 3	(1 – 5)	

- Внутри класса процесса Windows может назначить ~~изменить~~ потокам приоритет в диапазоне ± 2 от базового приоритета процесса
- Иногда ОС может повысить приоритет потока за предел базового уровня его процесса (не полностью исчерпав квант) или понизить приоритет. Это называется динамическим приоритетом. Поток не может опуститься ниже границы базового потока и выше предела класса (15).

7. Inter-process Communication

- Two fundamental models:



The End