Operating Systems
Processes and Scheduling

Jyrki Katajainen ⟨jyrki@di.ku.dk⟩

Source:

These slides are based on the slides provided by the authors of our textbook and further enhanced by Robert Glück.

Online exercise

How does `free` know how many bytes to free?

```
/* allocate n bytes from the heap */
void* malloc(unsigned int n);

/* release the block pointed to by p */
void free(void* p);
```

Online exercise

How much extra?

```
/* allocate n bytes from the heap */
void* malloc(unsigned int n);

/* release the block pointed to by p */
void free(void* p);
```

Benchmarking `malloc`

```c
#include <stdio.h>
#include <stdlib.h>

#define MEASURE(T, text) {\
  long long unsigned int previous = 0;\
  int i = 0;\
  printf("%s\t", text); \
  printf("%lu\t", sizeof(T));\
  for (; i < 11; ++i) {\
    T* p = (T*) malloc(sizeof(T));\
    long long unsigned int current = (long long unsigned int) p;\
    if (previous != 0) {\
      printf(" %llu", current - previous);\
    }\
    previous = current;\
  }\
  printf("\n");\
}
struct s1 {
  char c;
};

struct s24 {
  char c[24];
};

struct s25 {
  char c[25];
};

int main() {
  MEASURE(int, "int");
  MEASURE(struct s1, "s1");
  MEASURE(struct s24, "s24");
  MEASURE(struct s25, "s25");
  return 0;
}
```

```
jyrki@Janus$ gcc -std=c11 -Wpedantic -Wall space-model.c
jyrki@Janus$ ./a.out
int     4       32 32 32 32 32 32 32 32 32 32
s1      1       32 32 32 32 32 32 32 32 32 32
s24     24      32 32 32 32 32 32 32 32 32 32
s25     25      48 48 48 48 48 48 48 48 48 48
```

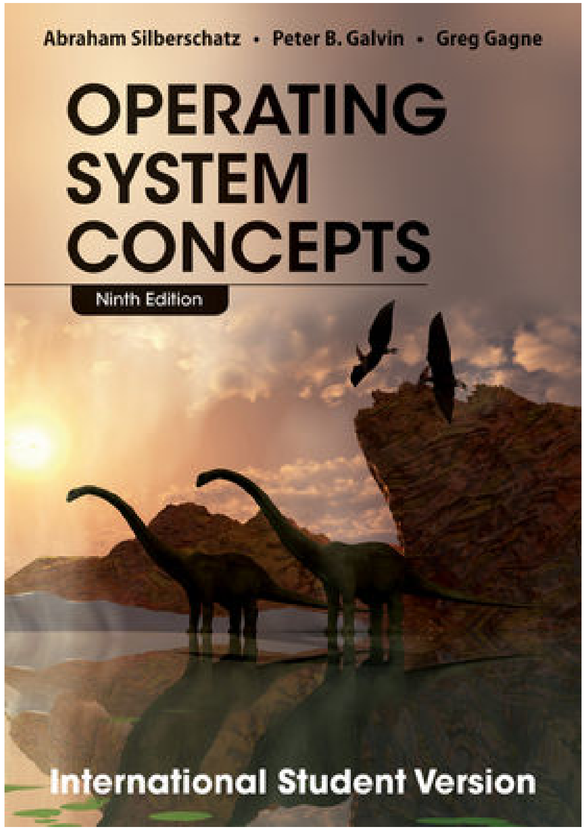
Today's Plan

- Processes:
  - Process concept
  - Context switch
  - Process family
  - Interprocess communication
- Scheduling:
  - How to utilize the system resources best?
  - How to ensure short waiting times?

Reading

- Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, *Operating System Concepts*, 9th Edition (2014)

  §§ 3 and 5



Motivation

This is just material you have to know to understand the (computer) world.

jyrki@Janus: ~/Courses/Operating-systems/Scheduling

```
top - 18:29:44 up 3 days,  3:21,  3 users,  load average: 0,15, 0,14, 0,26
Tasks: 224 total,   1 running, 223 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3,2 us,  0,9 sy,  0,0 ni, 95,8 id,  0,1 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   3974608 total,  3637608 used,   337000 free,   232776 buffers
KiB Swap:        0 total,        0 used,        0 free,   953896 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 1164 root      20   0  446m 101m  33m S  15,9  2,6 114:48.88 Xorg
16618 jyrki     20   0 2178m 1,0g  31m S   7,6 26,3 103:06.85 opera
 2084 jyrki     20   0 1258m  69m  17m S   3,6  1,8  61:12.90 compiz
 2199 jyrki     20   0  570m  92m 6160 S   2,3  2,4  44:57.72 skype
16651 jyrki     20   0 1030m  74m  20m S   1,3  1,9  97:18.91 opera:libflashp
 9113 jyrki     20   0  156m  51m  11m S   0,7  1,3   7:35.82 oald8-bin
   20 root      20   0     0    0    0 S   0,3  0,0   0:40.58 rcuos/2
 2025 jyrki     20   0 20500 1176  888 S   0,3  0,0   1:26.70 syndaemon
 2745 jyrki     20   0  649m  24m  13m S   0,3  0,6   3:20.55 gnome-terminal
28098 jyrki     20   0 28796 1716 1168 R   0,3  0,0   0:01.10 top
    1 root      20   0 27356 3068 1388 S   0,0  0,1   0:01.65 init
    2 root      20   0     0    0    0 S   0,0  0,0   0:00.02 kthreadd
    3 root      20   0     0    0    0 S   0,0  0,0   0:01.84 ksoftirqd/0
```

Process Management

Process: program in execution.
A unit of work in most computer systems.
To accomplish a task, a process needs resources:
CPU time, memory, files, I/O devices, etc.

System: collection of user and OS processes.
Traditionally: a process contains one thread of control; modern OS support multiple threads.

OS is responsible for process management:
process creation, deletion, communication, scheduling, synchronization, deadlock handling.

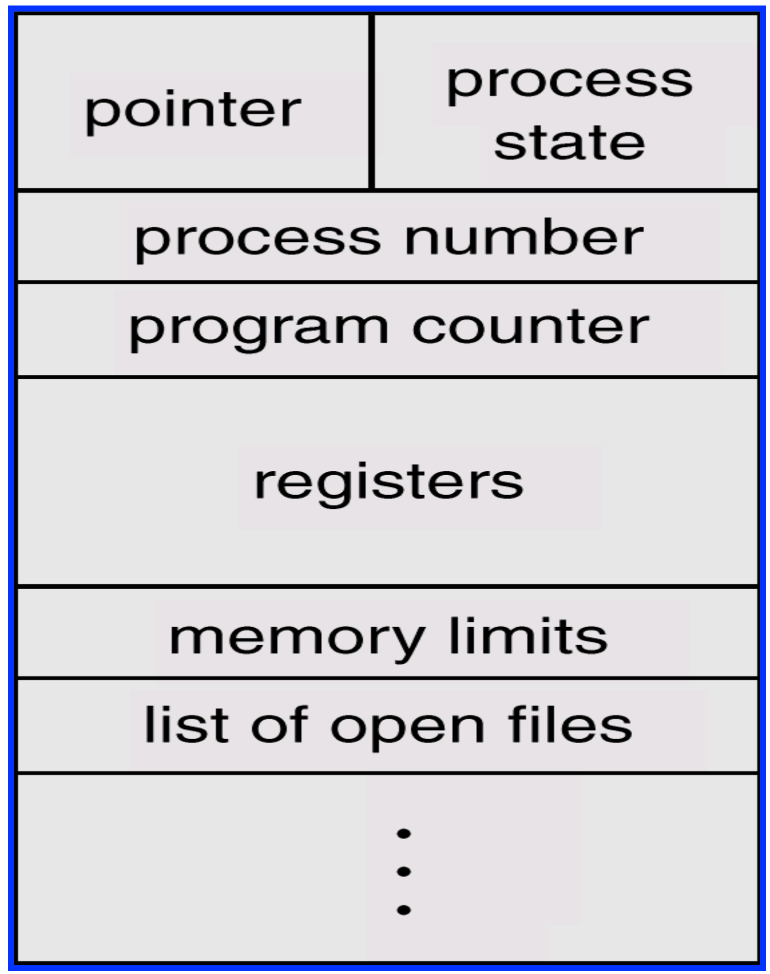
What is a Process?

- Process:
  - program code
  - state of execution (registers, open files, ...)
- Execution of process in sequential fashion
- Program functionality is specified by a series of instructions:
  - Executable code
  - High-level programming language
- Operating system executes a variety of user processes and system processes

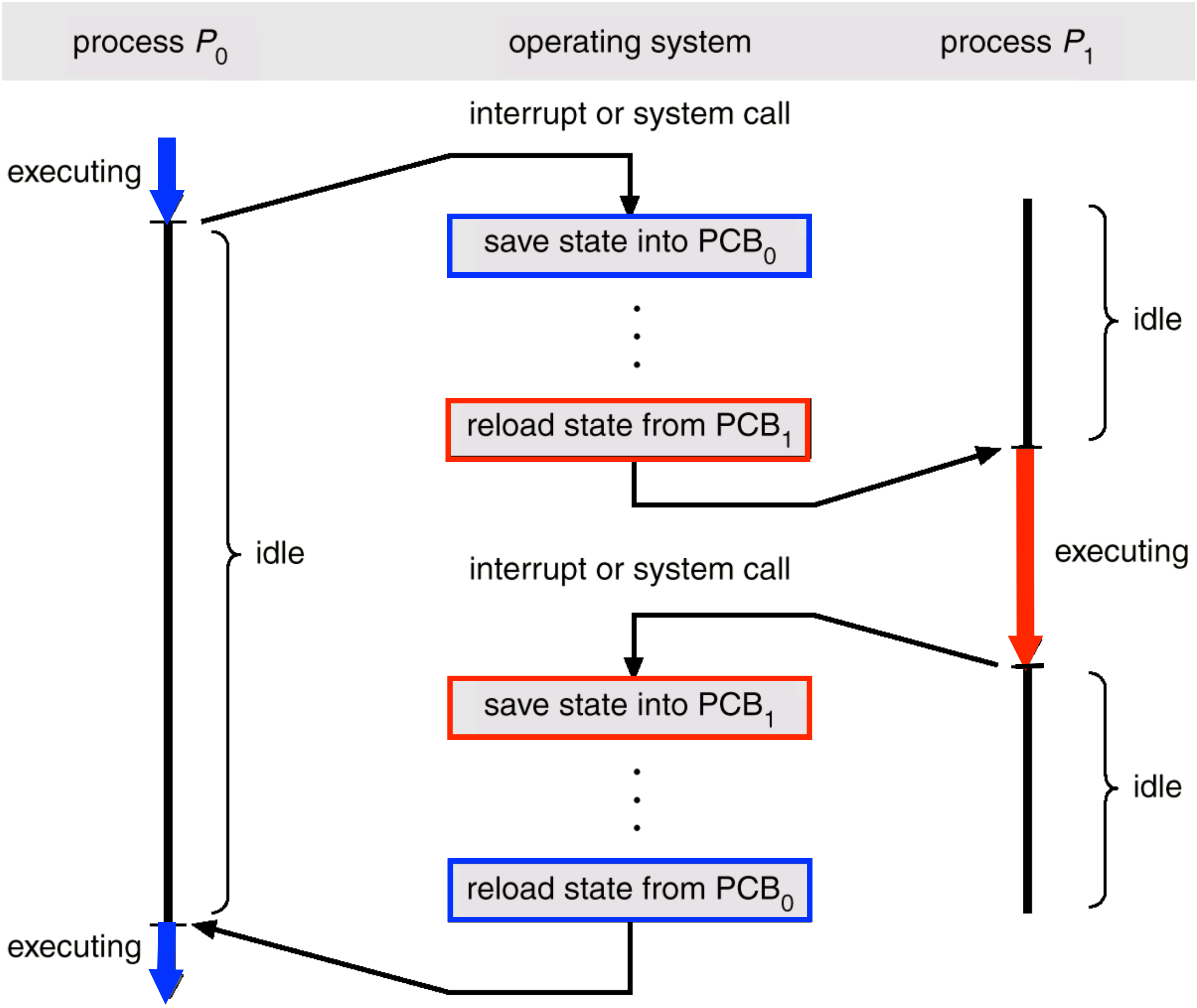
What’s in a Process?

- Information associated with each process:
  - Program code (same for all instances)
  - Program data
  - CPU registers
  - System resources:
    - Allocated memory
    - Open files, …
  - Accounting information:
    - Process ID
    - Resource usage, …

Process Control Block (PCB)



Switch from Process to Process



Context Switch

- Context switch is expensive (typically ≤10ms)
  - Direct costs:
    - Save & load execution state (CPU registers, counters,...)
    - Memory-management information (e.g., virtual memory)
  - Indirect costs:
    - CPU cache filled with data of old process
    - Other caches (e.g, hard disk) have similar problems

- Hardware supported context switch:
  - Single instruction to load and store registers
  - CPU has several register sets (e.g. 10 sets)
  - Intel hyperthreading (2 logical processors per CPU)

Life Cycle of a Process

As a process P executes, it changes state:

New: P is being created

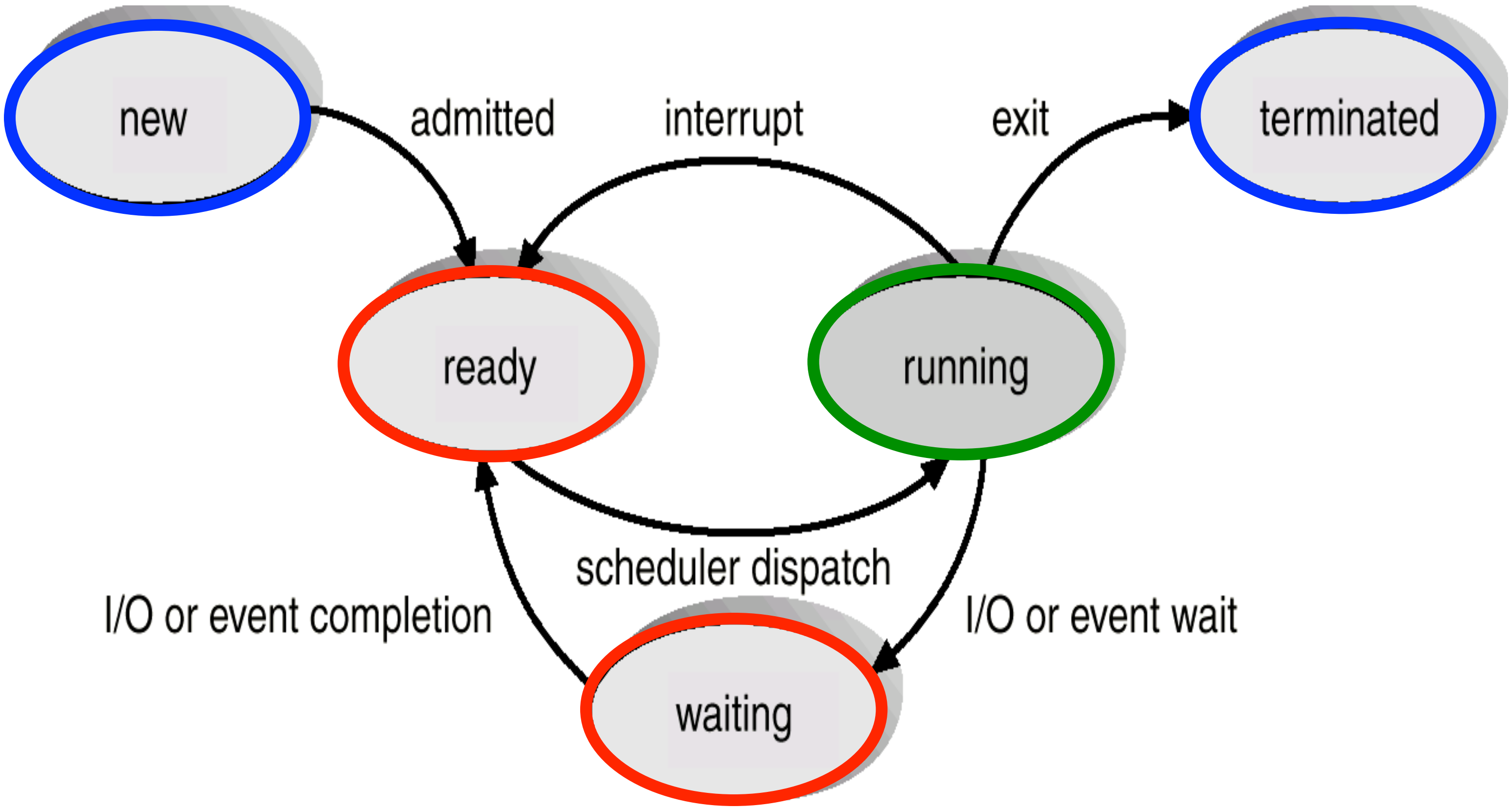
Ready: P is ready to be executed, but waits for CPU time

Running: P's instructions are executed

Waiting: P is waiting for some event

Terminated: P has finished execution

Process Life Cycle



Schedulers manage two queues (ready, waiting)

Scheduling Queues

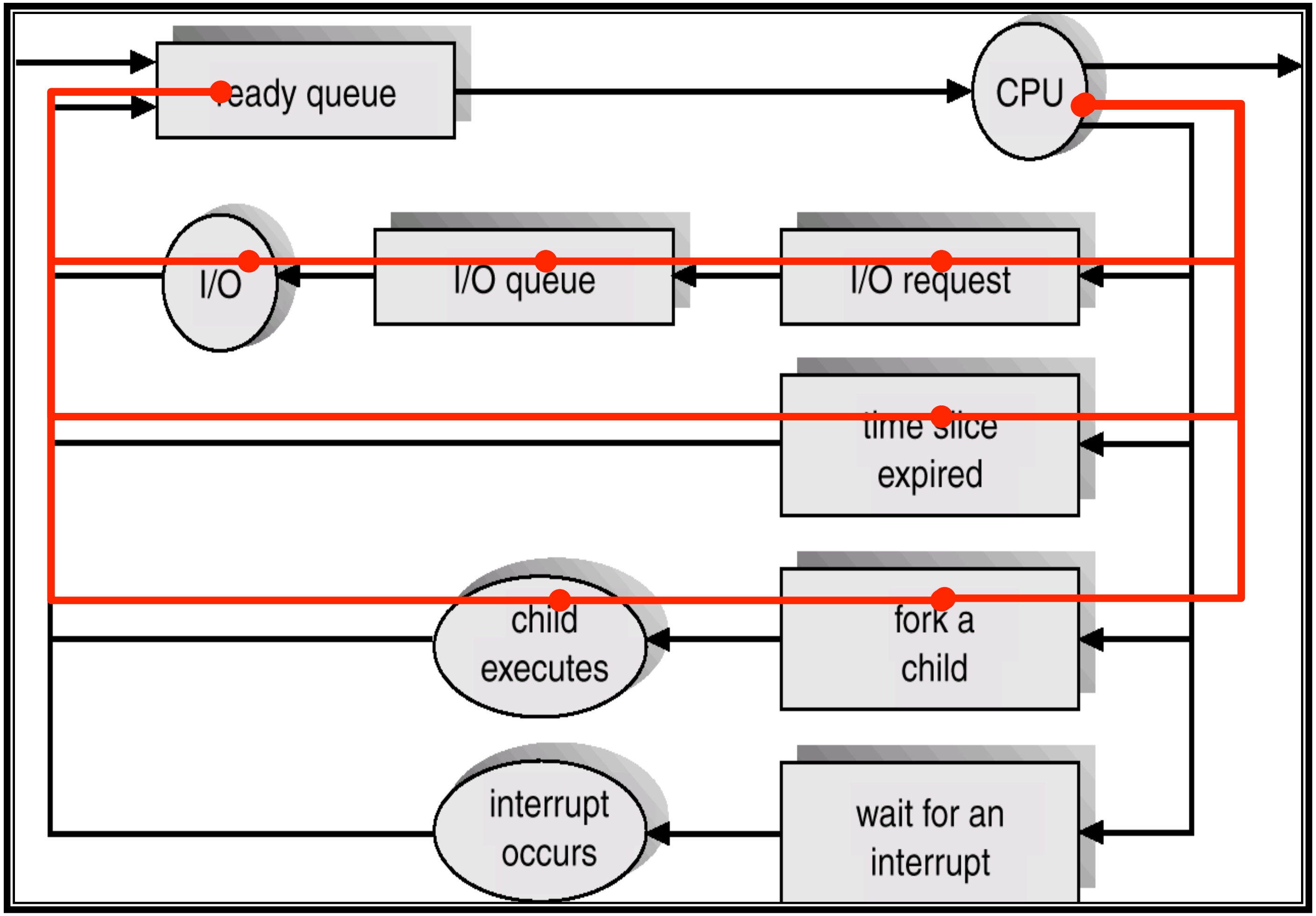
*Where are the processes that are **not running**?*

- **Ready queue**: ready and waiting for CPU.
- **Waiting queues**: waiting for an event:

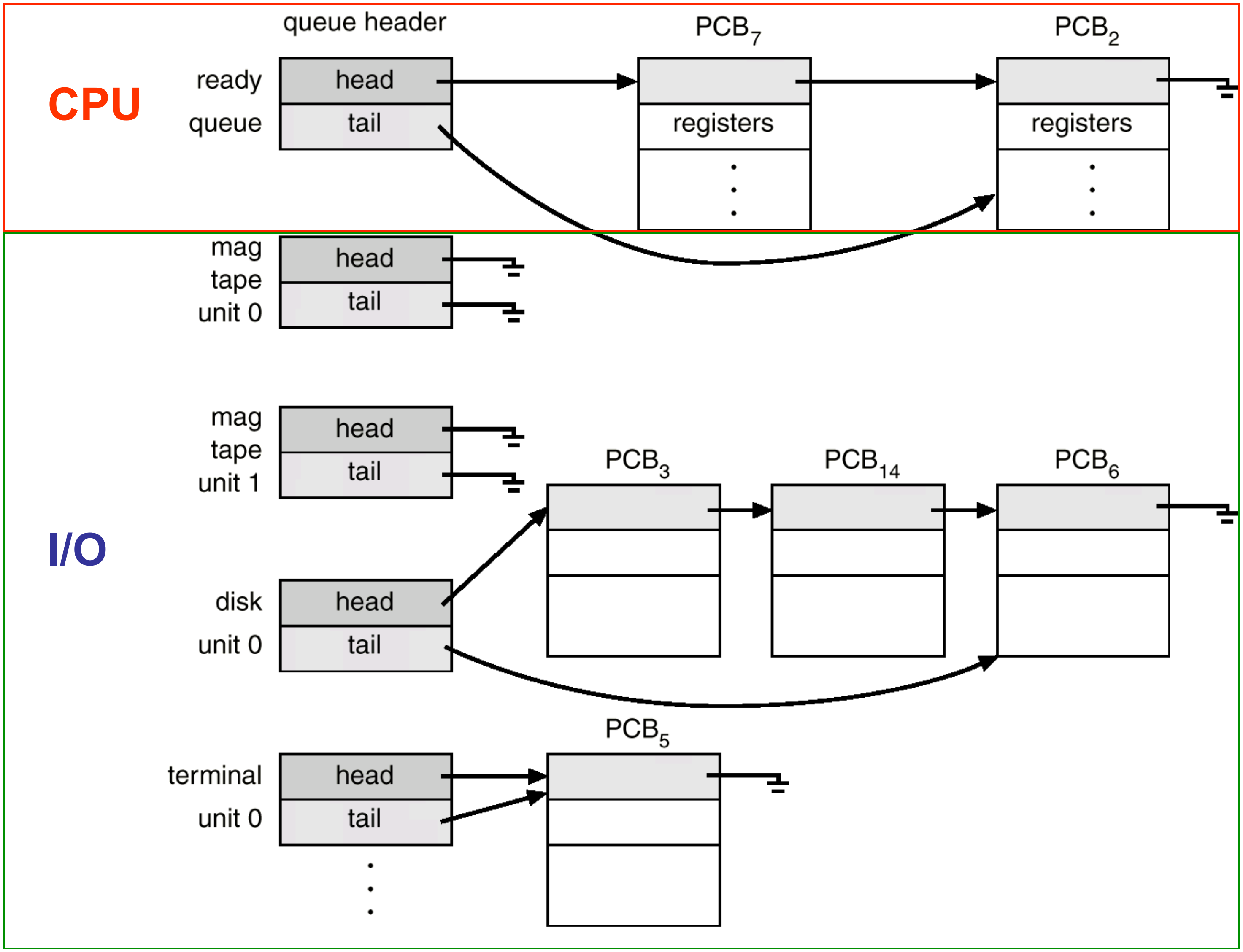
  I/O devices: waiting for an I/O device.

  Synchronization: limited access to a system resource requires wait.

Representation of Process Scheduling



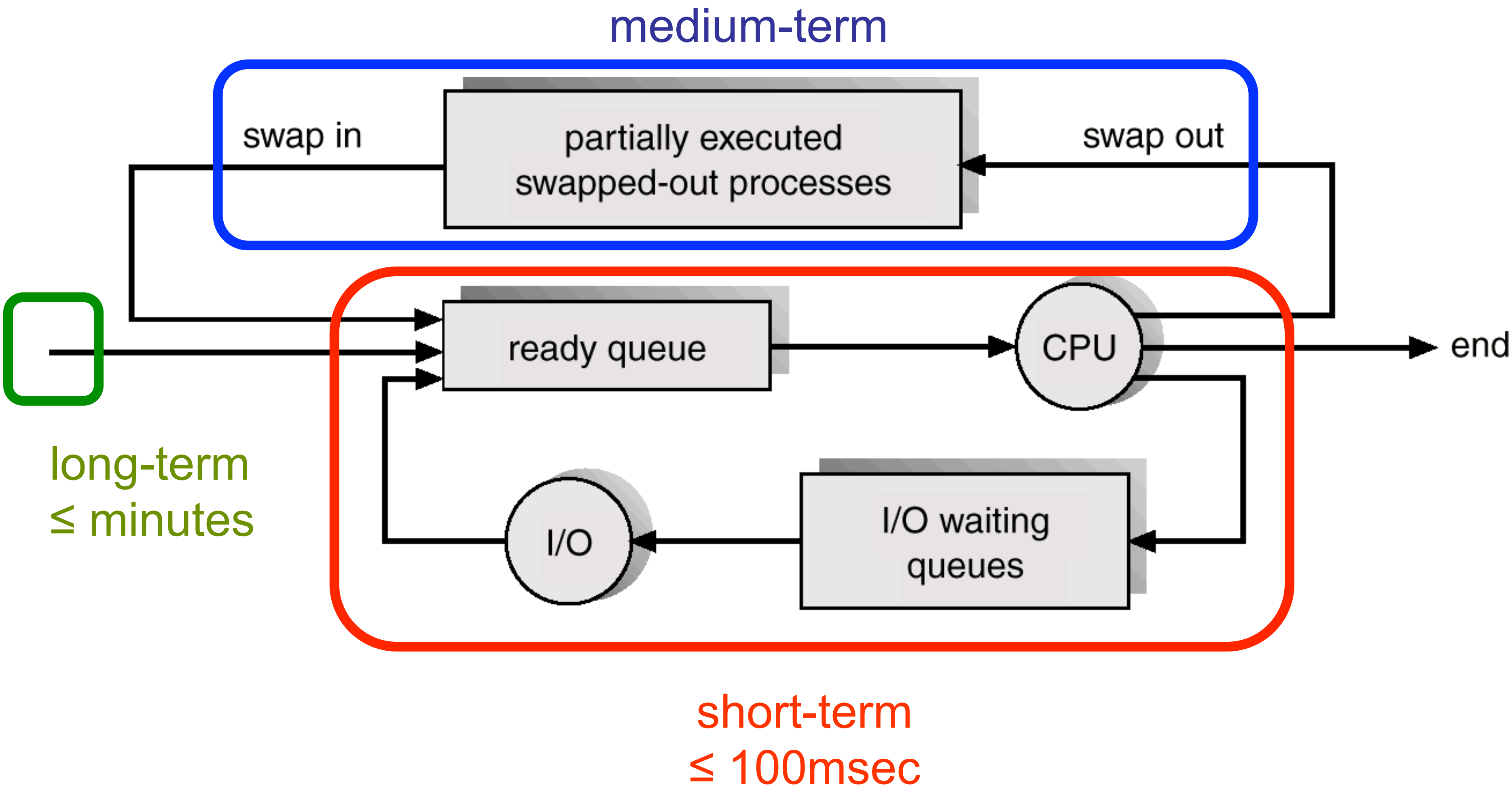
Several Scheduling Queues in OS



Schedulers

- **Short-term**: Selects which process should be executed next and allocates CPU.
- **Medium-term**: Process swap-in, swap-out, reduce degree of multiprogramming, improve process mix or when overcommitted.
- **Long-term**: Select which processes should be brought into the ready queue.

Scheduling Frequency



The Process Family

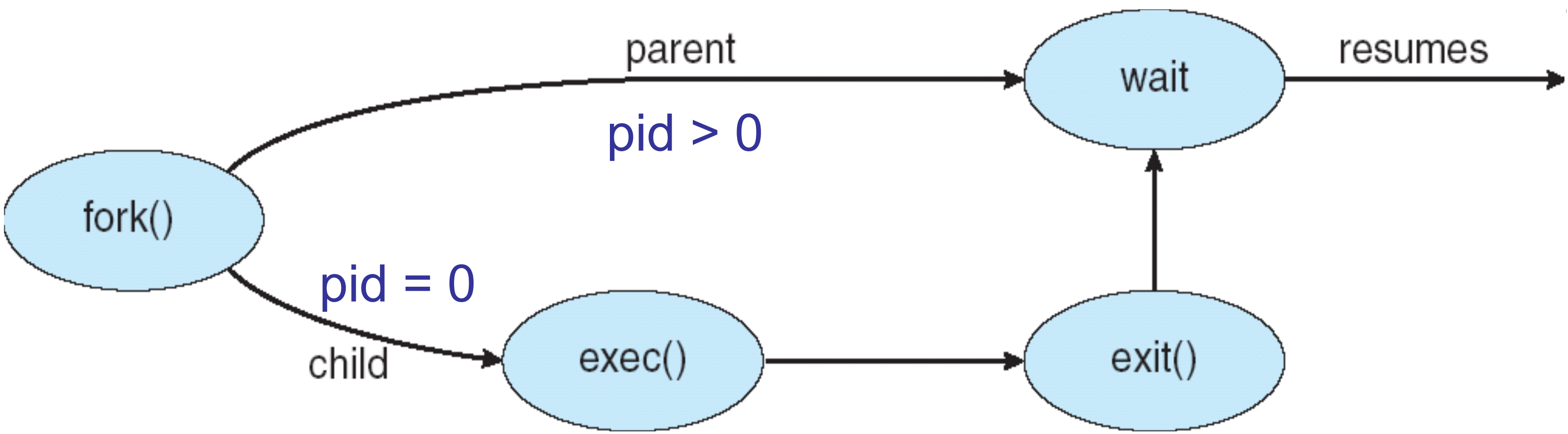
A process (**parent**) can start new processes (**children**):

- Child can inherit (parts of) the system resources of the parent:
  - Open files
  - Address space
- Child can be an instance of *another* program:
  - Typically, child inherits nothing from parent
- Execution of child:
  - Parent waits until child terminates
  - Parent and child execute concurrently
- When parent terminates:
  - Kill all children
  - Rights to children given to parent’s ancestor

UNIX Example

- `fork()` duplicate a process
  - Copy address space of parent
  - Run a copy of the *same* program
  - Return value:
    - **process_id** of child is returned to parent
    - **0** is returned to child
- `wait()` wait for termination of a child
  - Return value: **process_id** of child
- `exit()` terminate current process
- `execv(prog)` execute a new program **prog**

Process Creation by **`fork`**



Process creation:
in parent process: pid > 0
in child process: pid = 0

Process termination:
parent waits for child
to complete, then resumes

Forking a Separate Process in C

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
int pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
   }
   else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
   }
}
```

UNIX Example: command pstree

process tree

```
[cyller@localhost cyller]$ pstree
init-+-apmd
     |-atd
     |-battstat-applet
     |-bdflush
     |-bonobo-activati
     |-cardmgr
     |-crond
     |-cupsd---cups-polld
     |-devfsd
     |-esd
     |-gconfd-2
     |-gnome-panel
     |-gnome-settings-
     |-gnome-smproxy
     |-gnome-terminal-+-bash-+-gimp-+-screenshot
     |                |      |      `-script-fu
     |                |      `-pstree
     |                `-gnome-pty-helpe
     |-gpm
     |-gweather-applet---gweather-applet---gweather-applet
     |-kapmd
     |-keventd
     |-khubd
     |-kjournald
     |-klogd
     |-ksoftirqd_CPU0
     |-kswapd
     |-kupdated
     |-lisa
     |-mdrecoveryd
     |-metacity
     |-6*[mingetty]
     |-nautilus---nautilus---3*[nautilus]
     |-ntpd
     |-portmap
     |-prefdm---autologin---startx---xinit-+-X
     |                                     `-gnome-session
     |-rpc.statd
     |-saslauthd
     |-syslogd
     |-xfs
     |-xinetd---fam
     `-xscreensaver
```

UNIX Example: command ps

```
[cyller@localhost cyller]$ ps x -lH
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
100 S   501  1789  1247  0  69   0    -   667 do_sel ?            0:00 fam
100 S   501  1288  1125  0  69   0    -   597 wait4  ?            0:00 /bin/sh /usr/X11R6/bin/startx
000 S   501  1301  1288  0                           ?            0:00   xinit /etc/X11/xinit/xinitrc -- -deferglyphs 16
000 S   501  1339  1301  0                           ?            0:00     gnome-session
000 R   501  1817     1  1                           ?            0:03 /usr/bin/gnome-terminal
000 S   501  1818  1817  0  69   0    -   328 unix_s ?            0:00   /usr/lib/libzvt-2.0/gnome-pty-helper
000 S   501  1819  1817  0  75   0    -   685 wait4  pts/0        0:00   bash
000 R   501  1892  1819  0  77   0    -   747 -      pts/0        0:00     ps x -lH
000 S   501  1813     1  1  78   0    -  3419 do_pol ?            0:04 /usr/lib/battstat-applet-2 --oaf-activate-iid=OAFIID:GNOME_
000 S   501  1811     1  0  69   0    -  4114 do_pol ?            0:00 /usr/lib/gweather-applet-2 --oaf-activate-iid=OAFIID:GNOME_
040 S   501  1814  1811  0  69   0    -  4114 do_pol ?            0:00   /usr/lib/gweather-applet-2 --oaf-activate-iid=OAFIID:GNOM
040 S   501  1815  1814  0  69   0    -  4114 rt_sig ?            0:00     /usr/lib/gweather-applet-2 --oaf-activate-iid=OAFIID:GN
000 S   501  1802     1  0  69   0    -  8336 do_pol ?            0:01 nautilus --no-default-window --sm-client-id default3
040 S   501  1803  1802  0  69   0    -  8336 do_pol ?            0:00   nautilus --no-default-window --sm-client-id default3
040 S   501  1804  1803  0                                        0:00     nautilus --no-default-window --sm-client-id default3
040 S   501  1805  1803  0                                        0:00     nautilus --no-default-window --sm-client-id default3
040 S   501  1806  1803  0                                        0:00     nautilus --no-default-window --sm-client-id default3
040 S   501  1809  1803  0  69   0    -  8336 rt_sig ?            0:00     nautilus --no-default-window --sm-client-id default3
000 S   501  1800     1  0  70   0    -  4120 do_pol ?            0:00 gnome-panel --sm-client-id default2
000 S   501  1796     1  0  71   0    -  3082 do_pol ?            0:00 /usr/bin/metacity --sm-client-id=default1
000 S   501  1793     1  0  69   0    -  1048 do_sel ?            0:00 xscreensaver -nosplash
000 S   501  1791     1  0  69   0    -   408 do_sel ?            0:00 /usr/bin/esd -terminate -nobeeps -as 2 -spawnfd 22
000 S   501  1787     1  0  69   0    -  3653 do_pol ?            0:00 gnome-settings-daemon --oaf-activate-iid=OAFIID:GNOME_Setti
000 S   501  1785     1  0  69   0    -  1820 do_sel ?            0:00 gnome-smproxy --sm-client-id default0
000 S   501  1623     1  0  69   0    -  1007 do_pol ?            0:00 /usr/lib/bonobo-activation-server --ac-activate --ior-outpu
000 S   501  1493     1  0  69   0    -  1463 do_pol ?            0:00 /usr/lib/gconfd-2 12
[cyller@localhost cyller]$
```

One child

Four children

Cooperating Processes

1. **Independent process:** cannot affect or be affected by other processes.

2. **Cooperating process:** can affect or be affected by other processes.

Reasons for providing process cooperation:

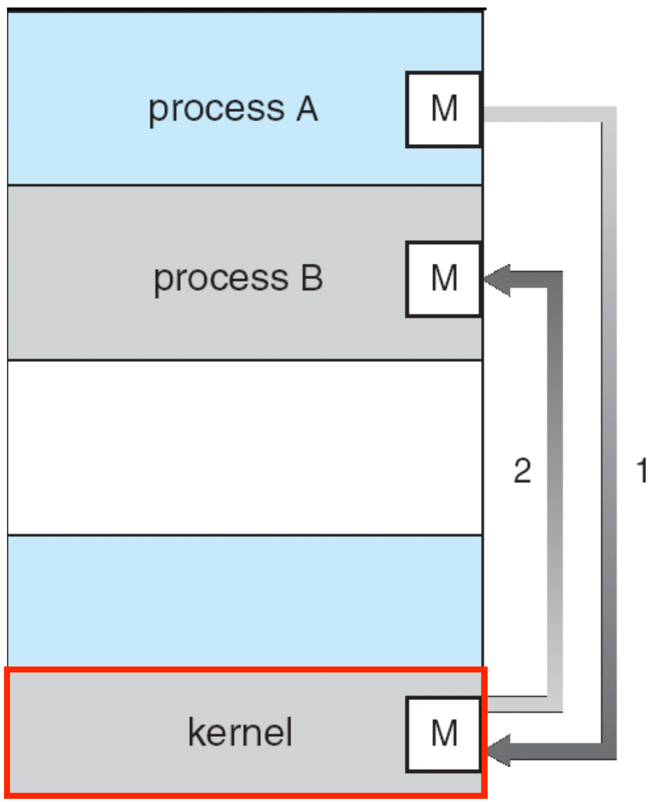
- **Information sharing:** for instance, share files
- **Computation speedup:** only if multiple processing elements (such as CPUs or I/O channels)
- **Modularity:** divide system functions into modules
- **Convenience**: several processes are natural for a task (e.g. edit, print, compile at the same time)

Interprocess Communication (IPC)

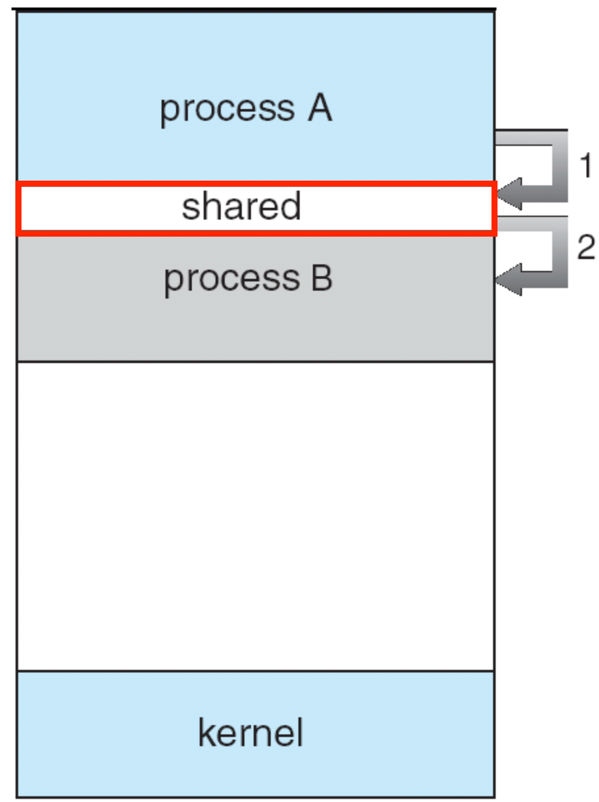
- Message passing:
  - Direct (process-to-process):
    - **send(451, message):** send msg to process with ID 451
    - **receive(id, message):** receive msg from any process
  - Indirect (via mailboxes):
    - Easier to establish many-to-many relations
    - Named mailboxes abstract from process ID
- Shared memory:
  - Shared OS memory: link input & output streams
    - UNIX pipe (example: **gunzip –v text.gz | less**)
  - Read/write to region of shared process memory
    - Harder to implement, larger amounts of data, faster

Two Communication Models

Message passing via kernel



Shared memory read / write

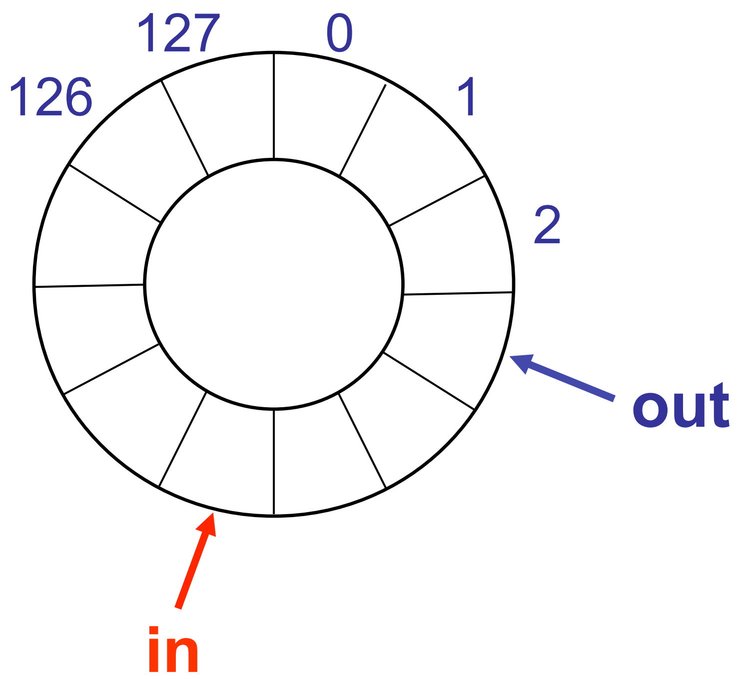


Producer-Consumer Problem

- Paradigm for cooperating processes: producer process produces data that is consumed by a consumer process
- Solution with shared-memory:
  - **circular buffer**: an array
  - **in**: pointer to next free entry
  - **out**: pointer to next available element

  Buffer:
  - unbounded = no practical limit on the buffer size
  - bounded = need to synchronize P’s and C’s speed

Circular Buffer



Wrap around array index:

…
(125 + 1) % 128 = 126
(126 + 1) % 128 = 127
(127 + 1) % 128 = 0
(0 + 1) % 128 = 1
…

buffer empty: out = in
buffer full: out = (in + 1) % 128

Circular Buffer: Implementation

Producer:

```
item nextProduced;

   while (1) {
        while (out == ((in + 1) % BUFFER_SIZE))
              yield(); /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
   }
```

buffer full condition

Consumer:

```
item nextConsumed;

   while (1) {
        while (out == in)
              yield(); /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
   }
```

buffer empty condition

CPU Scheduling

- Goal of CPU Scheduling:

  ”Best utilization of system resources”

- Different scheduling strategies:
  - First come, first served (FCFS)
  - Shortest job first (SJF)
  - Round robin (RR)
  - Priority based
  - Multilevel queue

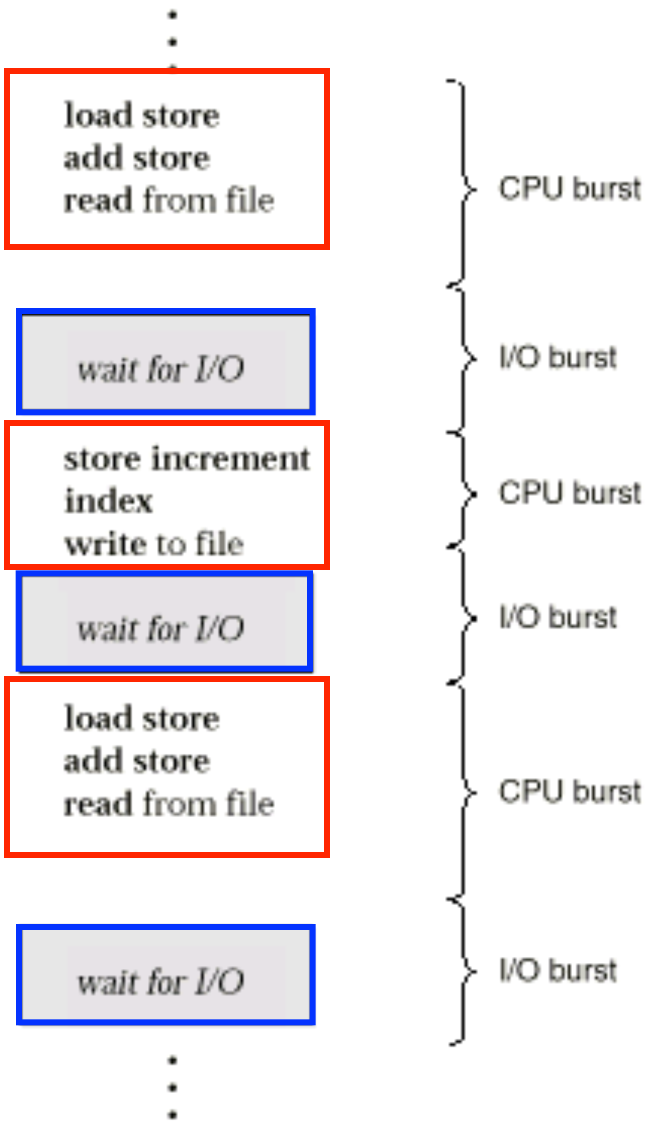
Goals of Scheduling

- Maximize utilization of system resources:
  - CPU
  - I/O devices
- Process execution consists of:
  - CPU execution
  - I/O requests
- Ensure low waiting times for user!

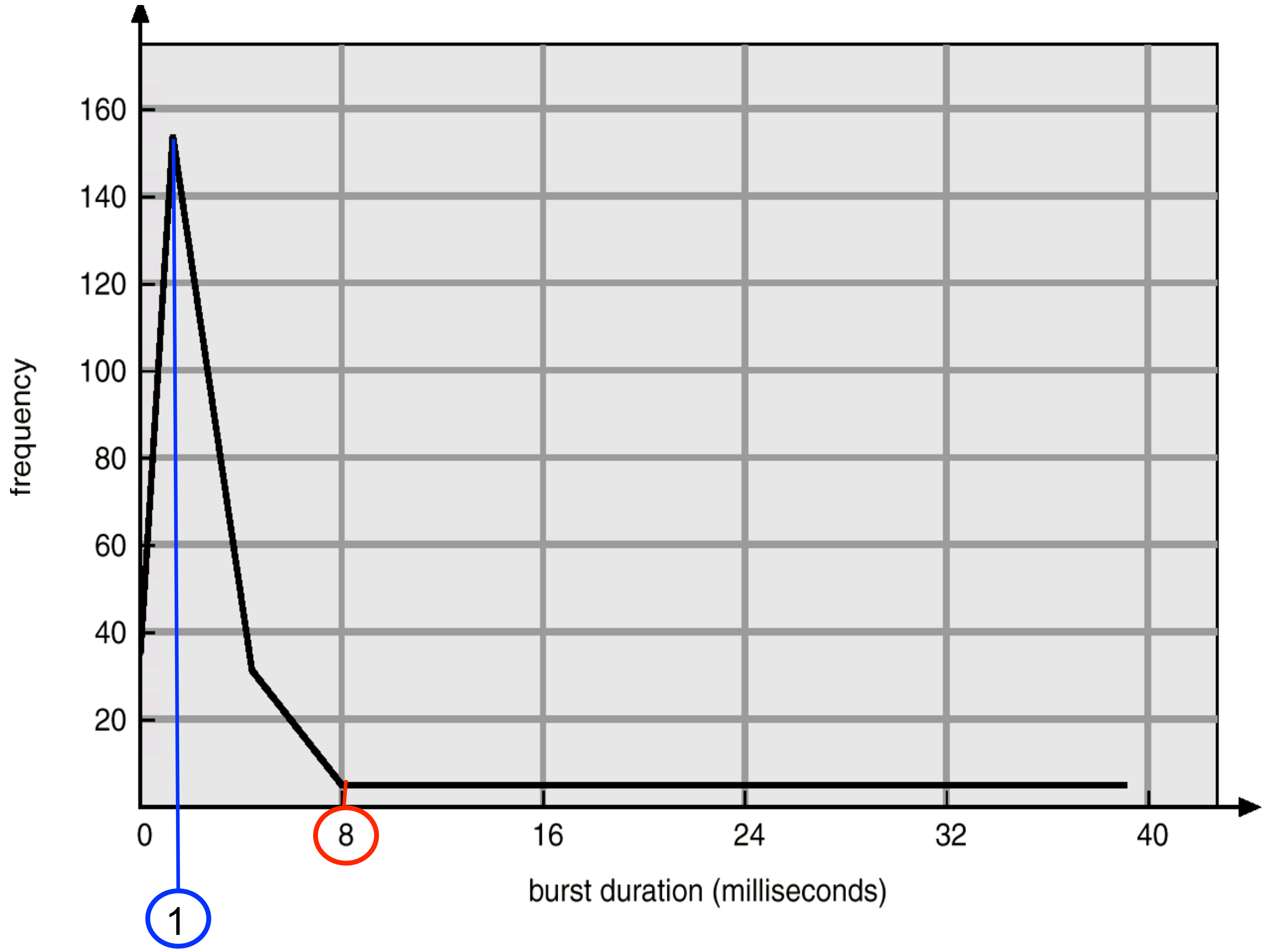
Usage of CPU versus I/O

- **CPU-bound**: uses CPU more than I/O devices

- **I/O-bound**: uses I/O devices more than CPU



Histogram of CPU-burst Times



CPU Scheduler

CPU scheduler selects a process for execution from a set of ready processes

Scheduling decisions may take place when:

1. **running** process switches to **waiting** state (I/O req)
2. process **terminates**
3. **running** process switches to **ready** state (interrupt)
4. **waiting** process switches to **ready** state (I/O done)

Nonpreemptive (voluntary switch): 1 & 2

Preemptive (forced switch): 3 & 4

Criteria for CPU Scheduling

- **CPU utilization:** Maximize
  - How much % of time CPU is busy (typical 40-90%)
- **Throughput:** Maximize
  - Number of processes that are completed per minute
- **Turnaround time:** Minimize
  - Time it takes to complete a process
- **Waiting time:** Minimize
  - Time spent waiting in the ready queue
- **Response time:** Minimize
  - Time it takes from a request to a response

First Come, First Served (FCFS)

Process	CPU burst time (msec)
P_1	24
P_2	3
P_3	3

- Processes arrive in the order: P_1 , P_2 , P_3
- **Gantt chart** for the schedule:

- **Waiting times**: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- **Average waiting time**: $(0+24+27)/3 = \mathbf{17}$

First Come, First Served

- Processes arrive in the order

$$P_2 \text{ , } P_3 \text{ , } P_1$$

- **Gantt chart** for the schedule:

- **Waiting times**: $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- **Average waiting time**: $(6+0+3)/3 = \mathbf{3}$

  $\Rightarrow$ Much better than previous case!

- **Convoy effect**: short processes behind long process

Shortest Job First (SJF)

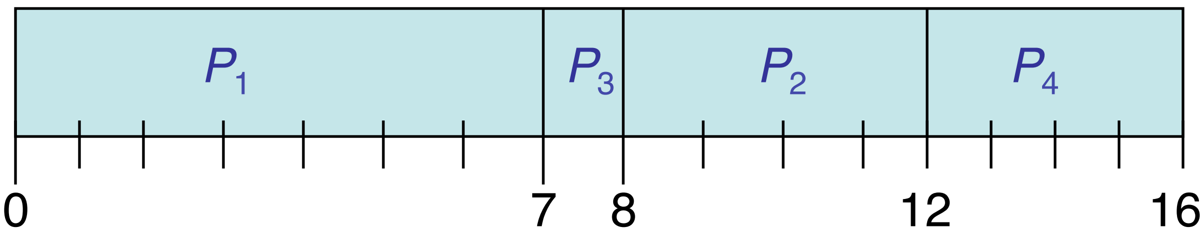
- CPU assigned to the process that has the shortest next CPU burst:
  - Nonpreemptive: running process is not interrupted
  - Preemptive: interrupt when a new process with a shorter CPU burst arrives at the ready queue

  Shortest waiting time for a set of processes!

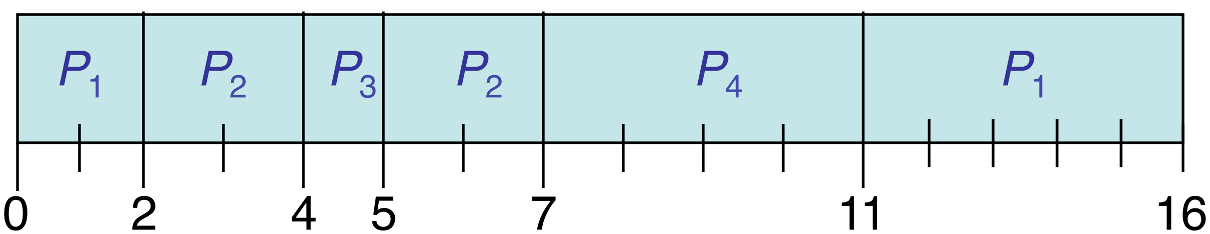
- Problem: need to know the burst time!
- Heuristics predict length of next burst time.

Example: Shortest Job First

- Nonpreemptive



- Preemptive



Process	P_1	P_2	P_3	P_4
Arrival	0	2	4	5
Burst	**7**	**4**	**1**	**4**

Average waiting times:

npre: $(0+6+3+7)/4 = \mathbf{4}$

pre: $(9+1+0+2)/4 = \mathbf{3}$

Predict Length of Next CPU Burst

Can be done by using the length of previous CPU bursts and **exponential averaging**:

1. t_n = actual length of n^{th} CPU burst
2. τ_n = predicted length of n^{th} CPU burst
3. τ_{n+1} = predicted length of next CPU burst
4. $0 \leq \alpha \leq 1$
5. Define: $\boxed{\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n}$

(α controls the relative weight of recent (t_n) and past (τ_n) bursts)

Exponential Averaging

We expand the formula:

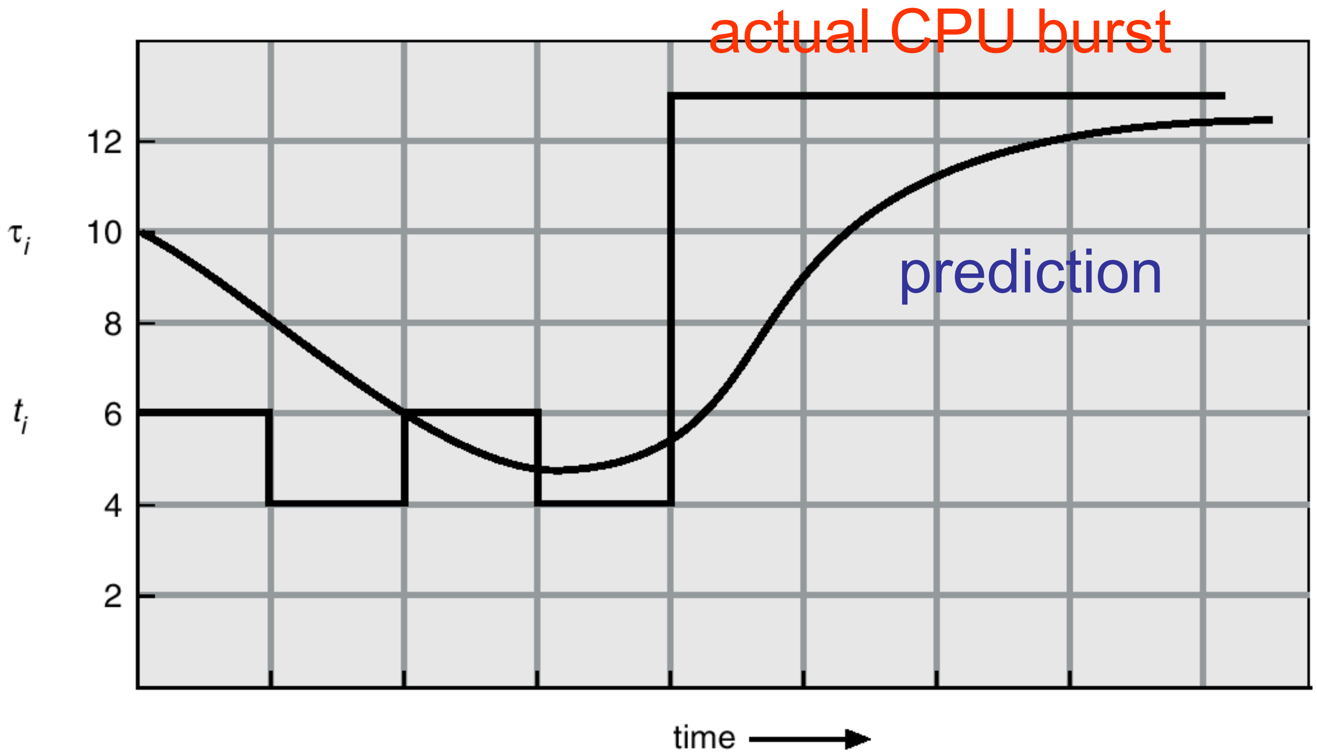
$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\, \alpha\, t_{n-1} + \ldots + (1-\alpha)^n\, \alpha\, t_1 + (1-\alpha)^{n+1}\, \tau_0$$

Special cases:

- α=0: $\tau_{n+1} = \tau_n$ recent CPU burst does *not* matter
- α=1: $\tau_{n+1} = t_n$ only the recent CPU burst matters

- Because α, $(1 - \alpha) \leq 1$, each sucessive term (right) has less weight than its predecessor (left).

Example: Prediction of Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

Exponential average with α=1/2 τ_0=10.

Round Robin (RR)

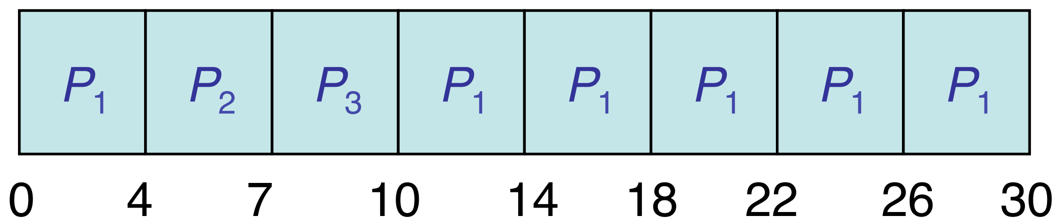
- Each process gets a small unit of CPU time, called **time quantum**, typically q = 10-100 ms.
- After q has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue, then each process gets $1/n$ of the CPU time, and no process waits longer that $(n-1)q$ time.
- Performance:
  - q large $\Rightarrow$ long waiting times (same as FCFS)
  - q small $\Rightarrow$ q must be large enough with respect to context-switch time; otherwise the overhead is too high

Example: Round Robin

- Process CPU Burst Times:

  $P_1 = 24$, $P_2 = 3$, $P_3 = 3$

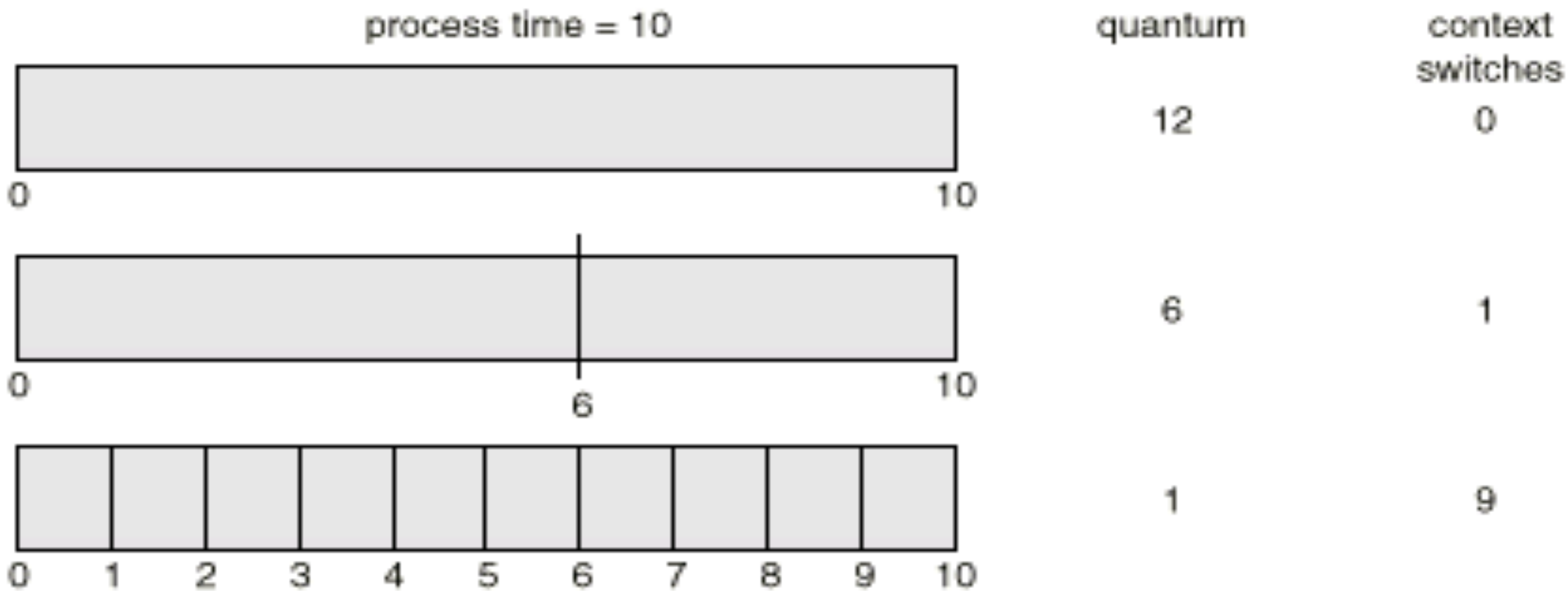
- **Gantt chart** (time quantum = 4 msec):



- **Average waiting time:** ((10-4)+4+7)/3 = **5.66**
- Typically, higher average turnaround than SJF, but *better response* ⇒ *time sharing systems*

Time Quantum and Context Switch Time

Smaller time quantum $\Rightarrow$ more context switches!



Rule of thumb: context switch time ≤ 10% of time quantum, 80% of CPU bursts should be shorter than time quantum.

Time quantum typically 10-100 ms.

Time quantum too large: degenerates to FCFS.

Priority Scheduling

- A priority number is associated with each process. Example: 0 (high) - 7 (low)
- The CPU is allocated to the process with the highest priority (equal priority in FCFS)
- SJF does priority scheduling:
  priority = the predicted next CPU burst time

- Problem: Starvation – low priority processes may never execute
- Solution: Aging – as time progresses, increase the priority of the processes

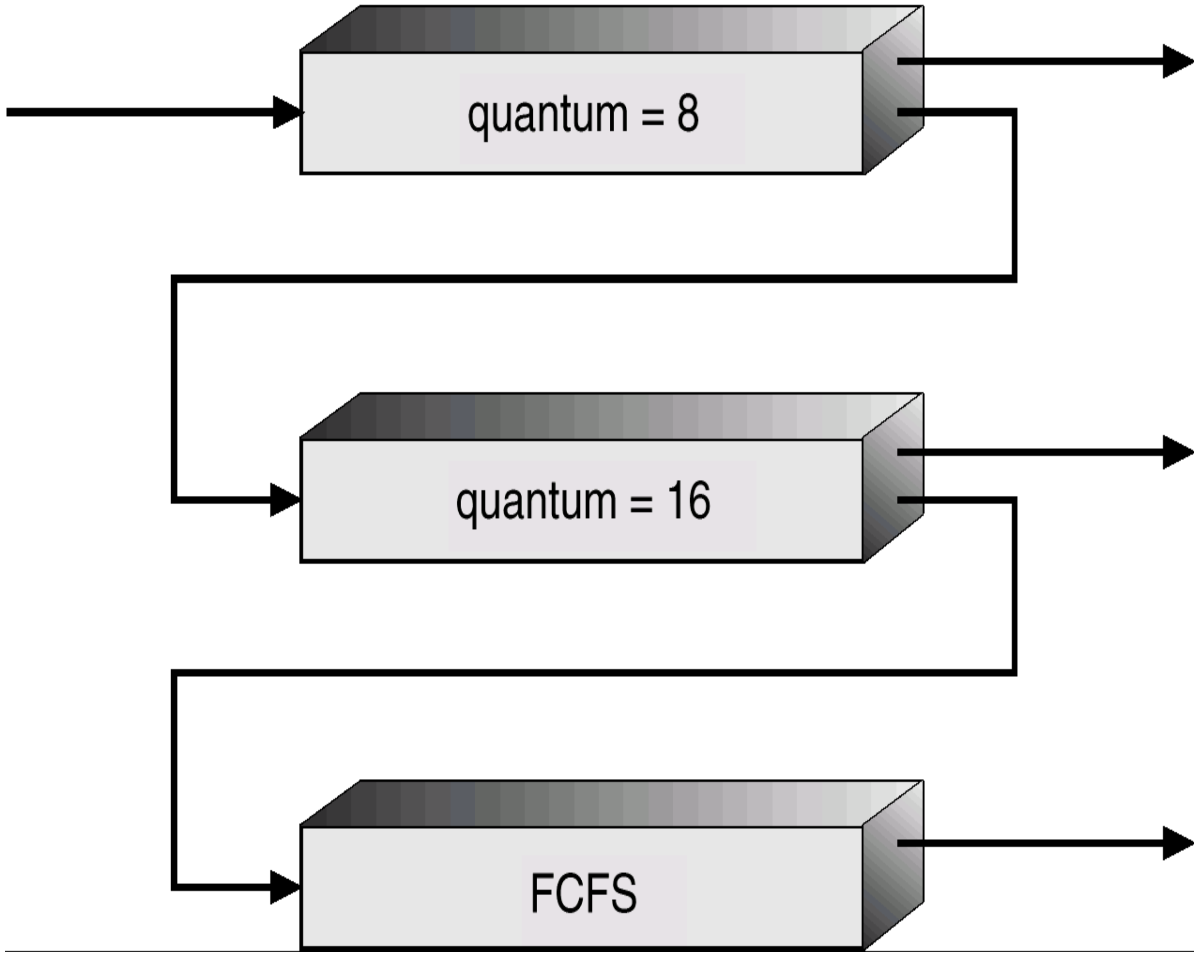
Multilevel Ready Queue

- Queue is partitioned into separate queues:
  - foreground queue (interactive)
  - background queue (batch)
- Each queue has its own scheduling algorithm:
  - foreground: RR
  - background: FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling: serve all from foreground, then from background. Possibility of starvation.
  - Time slice: example: 80% to foreground in RR, 20% to background in FCFS.

Multilevel Feedback Queue

- A process can move between the various queues; e.g. implement **aging**.
- Observe processes and place them into queue according to their behavior:
  - A foreground process that often (or once!) exceeds its time quantum is placed into the background queue

Example of Multilevel Feedback Queue



Summary

- Process: program in execution, process switch, process queues
- Threads: parallelism within a process
- Scheduling: maximize CPU utilization, minimize waiting times, forced vs. voluntary process switch, scheduling strategies