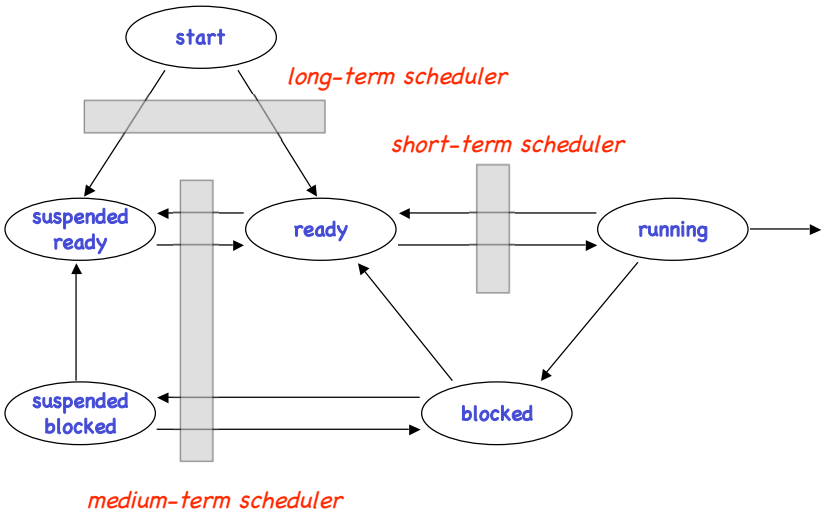
CPU Scheduling

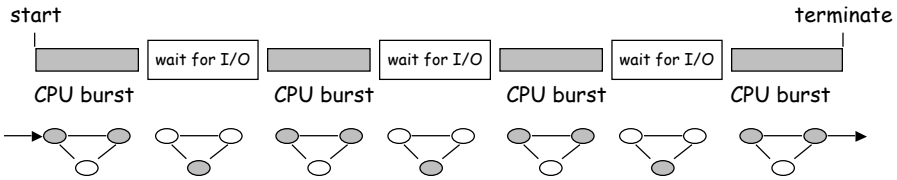
- Schedulers in the OS
- Structure of a CPU Scheduler
  - Scheduling = Selection + Dispatching
- Criteria for scheduling
- Scheduling Algorithms
  - FIFO/FCFS
  - SPF / SRTF
  - Priority / MLFQ
- Thread Dispatching (hands-on!)

Schedulers



Short-Term Scheduling

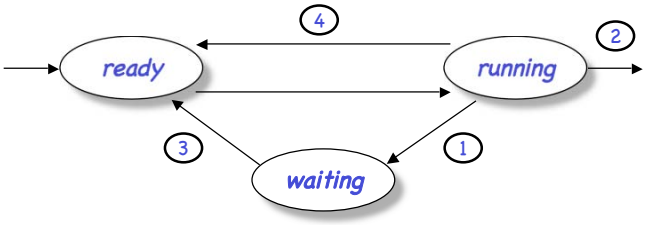
- Recall: Motivation for multiprogramming -- have multiple processes in memory to keep CPU busy.
- Typical execution profile of a process/thread:



- CPU scheduler is managing the execution of CPU bursts, represented by processes in ready or running state.

Scheduling Decisions

"Who is going to use the CPU next?!"



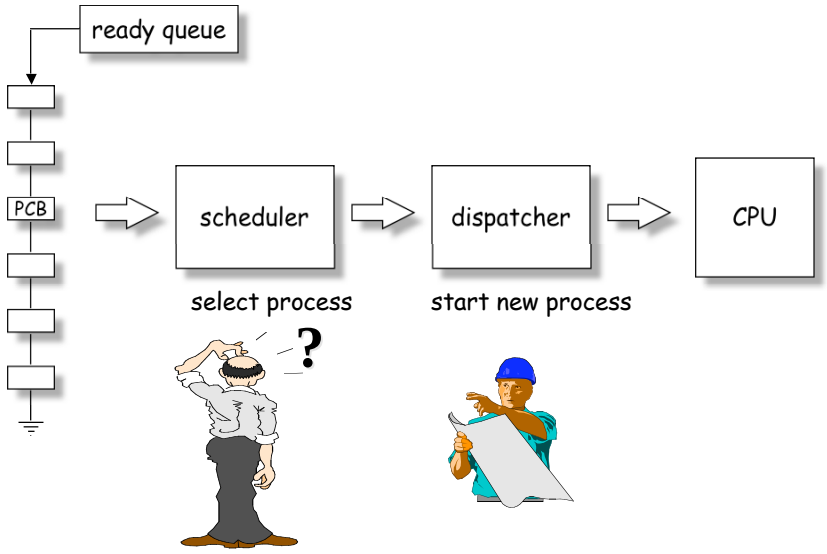
non-preemptive

Scheduling decision points:

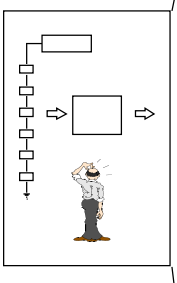
- 1. The running process changes from *running* to *waiting* (current CPU burst of that process is over).
- 2. The running process terminates.
- 3. A waiting process becomes *ready* (new CPU burst of that process begins).
- 4. The current process switches from *running* to *ready* .

preemptive

Structure of a Scheduler

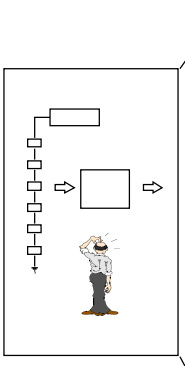


What Is a Good Scheduler? Criteria



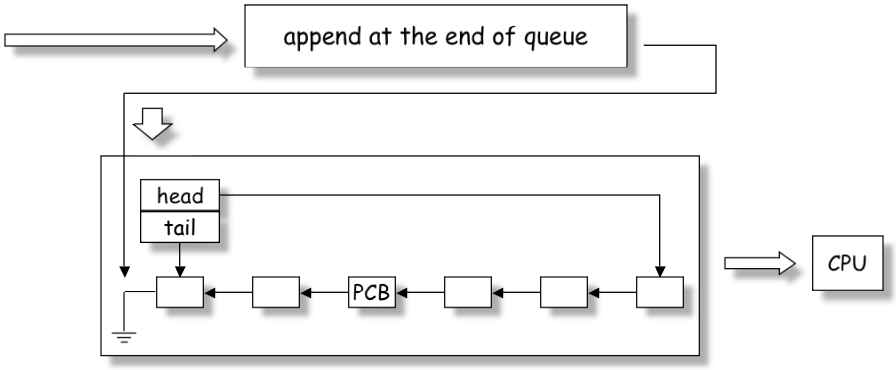
- User oriented:
  - **Turnaround time** : time interval from submission of job until its completion
  - **Waiting time** : sum of periods spent waiting in ready queue
  - **Response time** : time interval from submission of job to first response
  - **Normalized turnaround time**: ratio of turnaround time to service time
- System oriented:
  - **CPU utilization** : percentage of time CPU is busy
  - **Throughput** : number of jobs completed per time unit
- Any good scheduler should:
  - *maximize* CPU utilization and throughput
  - *minimize* turnaround time, waiting time, response time
- Huh?
  - maximum/minimum values *vs.* average values *vs.* variance

Scheduling Algorithms



- FCFS : First-come-first-served
- SPN: Shortest Process Next
- SRT: Shortest Remaining Time
- priority scheduling
- RR : Round-robin
- MLFQ: Multilevel feedback queue scheduling
- Multiprocessor scheduling

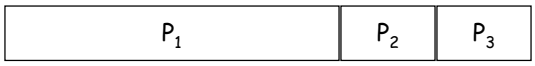
First-Come-First-Served (FCFS/FIFO)



- Advantages:
  - very simple
- Disadvantages:
  - long average and worst-case waiting times
  - poor dynamic behavior (convoy effect)

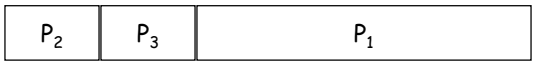
Waiting Times for FCFS/FIFO

- Example: $P_1 = 24$, $P_2 = 6$, $P_3 = 6$



$W_{awg} = (24+30)/3 = 18$
$W_{wc} = 30$

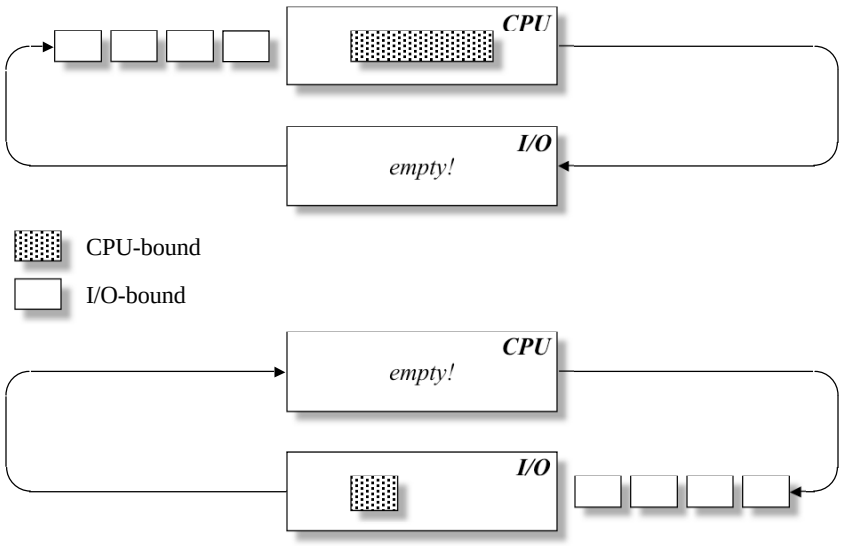
Different arrival order:



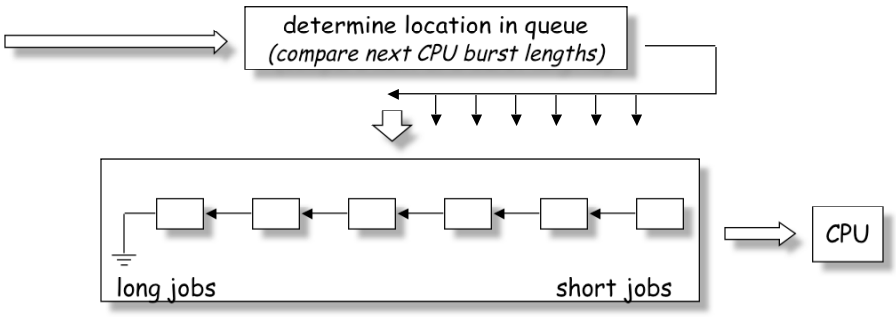
$W_{awg} = (6+12)/3 = 6$
$W_{wc} = 12$

- Average waiting times is not minimal.
- Waiting times may substantially vary over time.
- Worst-case waiting times can be very long.

Convoy Effects



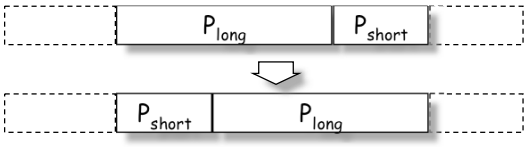
Shortest Process Next



- Whenever CPU is idle, picks process with shortest next *CPU burst*.
- Advantages: minimizes average waiting times.
- Problem: How to determine length of next CPU burst?!
- Problem: Starvation of jobs with long CPU bursts.

SJF Minimizes Average Waiting Time

- Provably optimal: Proof: swapping of jobs



$dW = t_{short} - t_{long} < 0$

- Example:

6	12	8	4	W = 6+18+26 = 50
6	8	12	4	W = 6+14+26 = 46
6	8	4	12	W = 6+14+18 = 38
6	4	8	12	W = 6+10+18 = 34
4	6	8	12	W = 4+10+18 = 32

SJF in Practice ?

How to determine execution time of next CPU burst ?!

- wild guess?
- code inspection?

- Forecasting (i.e. estimation)

$$S_{n+1} = F(T_n, T_{n-1}, T_{n-2}, T_{n-3}, T_{n-4}, \ldots)$$

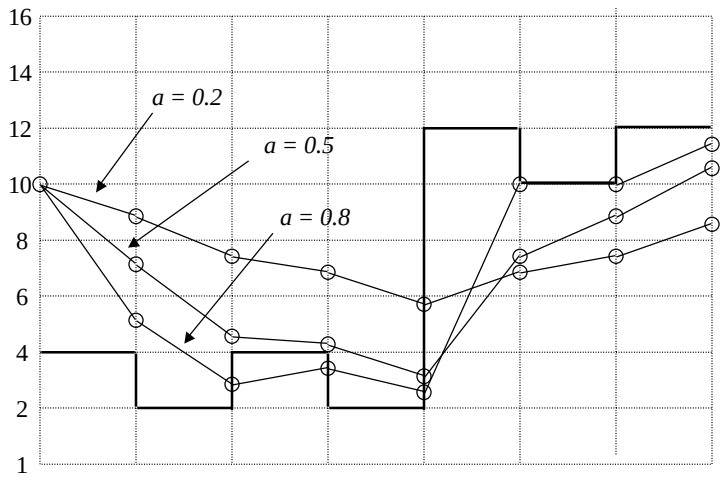
- Simple forecasting function: exponential average:

$$S_{n+1} = a\ T_n + (1-a)\ S_n$$

- Example: $a = 0.8$

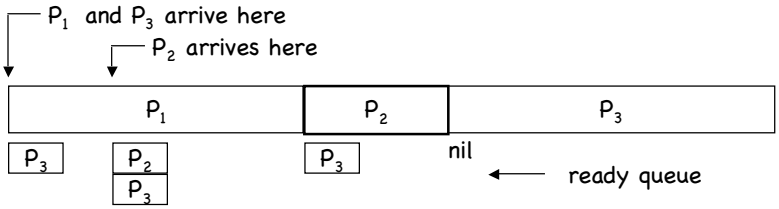
$$S_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032T_{n-2} + 0.0064T_{n-3} + \ldots$$

Exponential Averaging: Example

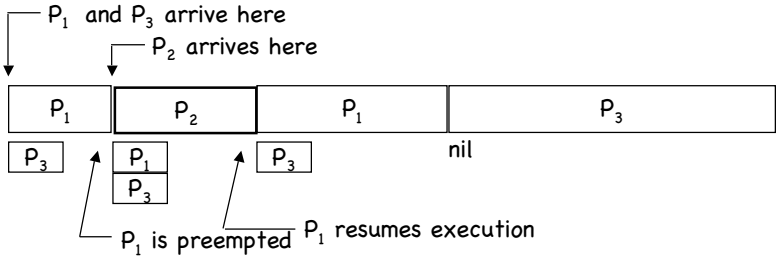


Preemptive SPN: Shortest-Remaining-Time-First

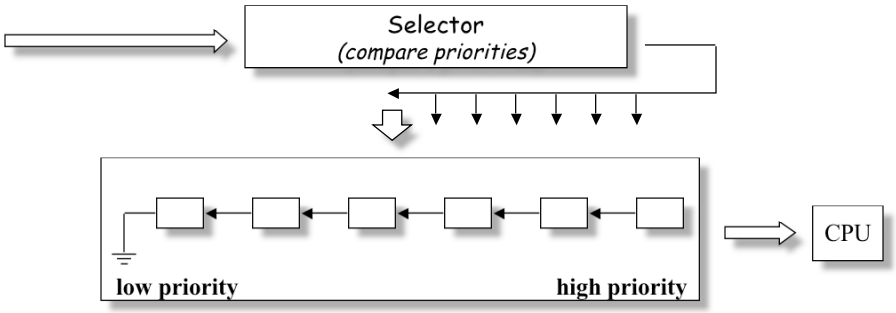
- SPN:



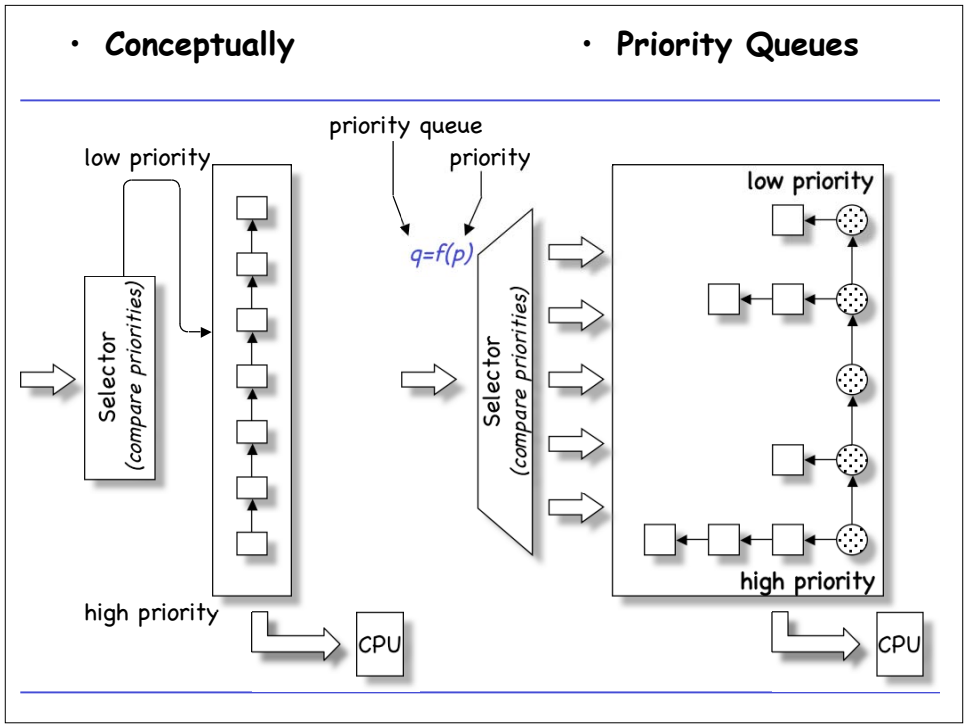
- SRT:

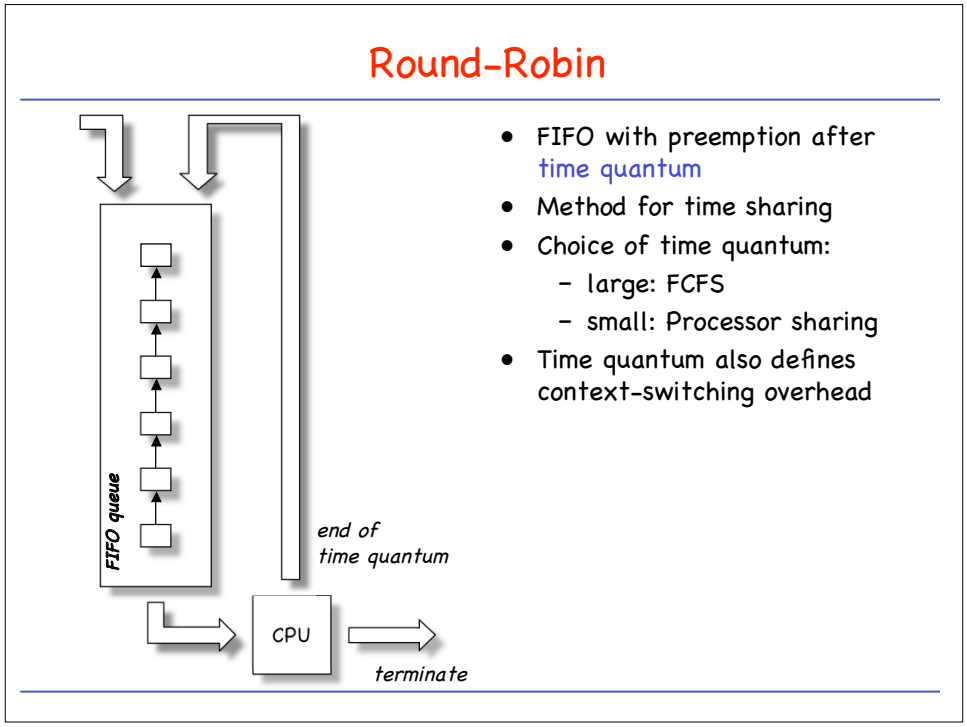


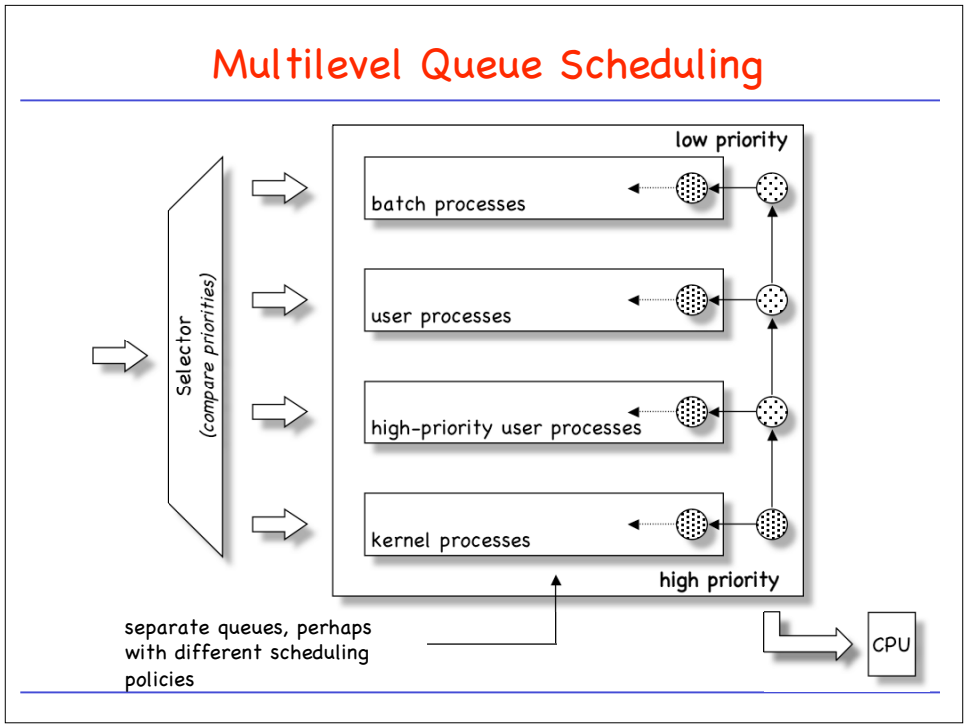
(Fixed) Priority Scheduling

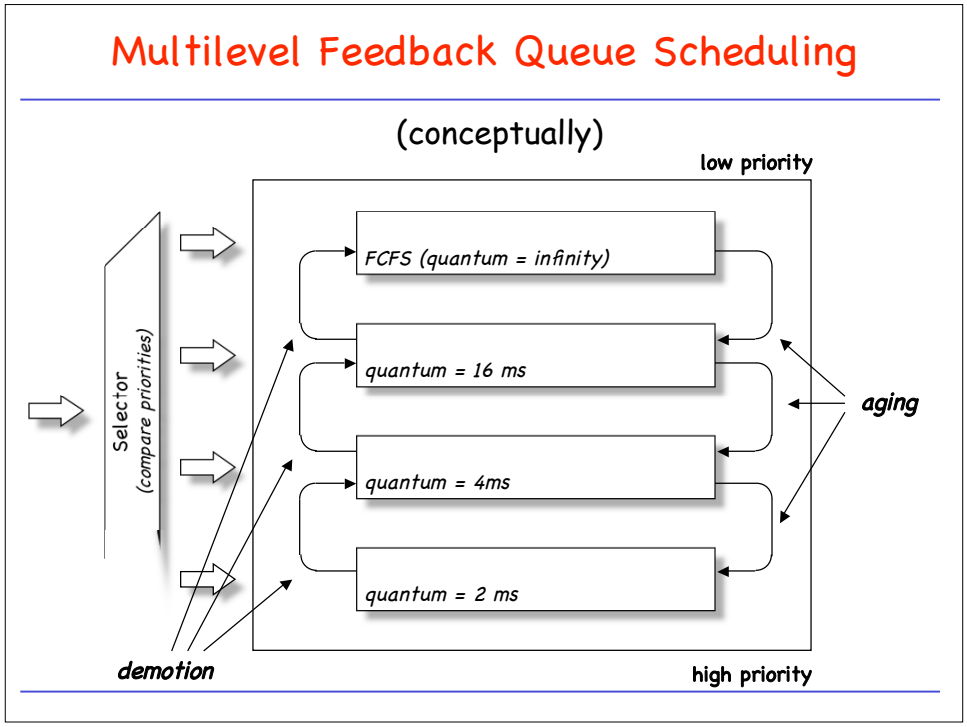


- Whenever CPU is idle, picks process with highest priority.
- Priority:
  - process class, urgency, pocket depth.
- Unbounded blocking: Starvation
  - Increase priority over time: aging


• Conceptually
• Priority Queues
low priority
Selector
(compare priorities)
high priority
CPU
priority queue
priority
q=f(p)
Selector
(compare priorities)
low priority
high priority
CPU

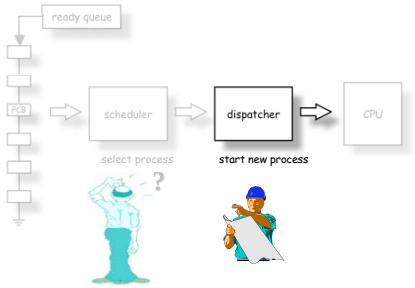

Round-Robin
FIFO queue
end of
time quantum
CPU
terminate
• FIFO with preemption after time quantum
• Method for time sharing
• Choice of time quantum:
– large: FCFS
– small: Processor sharing
• Time quantum also defines context-switching overhead


Multilevel Queue Scheduling
low priority
batch processes
user processes
high-priority user processes
kernel processes
high priority
Selector
(compare priorities)
separate queues, perhaps
with different scheduling
policies
CPU


Multilevel Feedback Queue Scheduling
(conceptually)
low priority
FCFS (quantum = infinity)
quantum = 16 ms
quantum = 4ms
quantum = 2 ms
Selector
(compare priorities)
aging
demotion
high priority

CPU Scheduling

- Schedulers in the OS
- Structure of a CPU Scheduler
  - Scheduling = Selection + Dispatching
- Criteria for scheduling
- Scheduling Algorithms
  - FIFO/FCFS
  - SPF / SRTF
  - Priority / MLFQ
- Thread Dispatching (hands-on!)



Managing and Dispatching Threads (1)

```
typedef enum {THRD_INIT, THRD_READY, THRD_SUSPENDED, THRD_RUNNING,
              THRD_EXIT, THRD_STOPPED} THREAD_STATE;

typedef struct thread_context {
  reg_t    s0, s1, s2, s3;
  reg_t    s4, s5, s6, s7;
  reg_t    gp;
  reg_t    ra;
  reg_t    fp;
  reg_t    sp;
  reg_t    pc;
} THREAD_CONTEXT;
```

```
class Thread : public PObject {
protected:
  char               name[15];
  Addr               stack_pointer;
  friend class Scheduler;
  THREAD_CONTEXT     thread_context;
  THREAD_STATE       thread_state;
  Scheduler        * sched; /* pointer to global scheduler */
public:
  Thread(char        _name[],
         int         (*_thread_func_addr)(),
         int         _stack_size,
         Scheduler * _s);
  ~Thread();
  /* -- THREAD EXECUTION CONTROL */
  virtual int start() {
    /* Start thread and toss it on the ready queue. */
    sched->resume();
  }
  virtual int kill() {
    /* Terminate the execution of the thread. */
    sched->terminate();
  }
};
```

Managing and Dispatching Threads (2)

```
class Scheduler {
private:
  int yield_to(Thread * new_thread);   /* Calls low-level dispatching mechanisms. */
protected:
  Thread * current_thread;
  /* -- MANAGEMENT OF THE READY QUEUE */
  virtual int remove_thread(Thread * _thr) {}; /* = NULL; */
  /* Remove the Thread from any scheduler queues. */
  virtual Thread * first_ready() {}; /* = NULL;*/
  /* Removes first thread from ready queue and returns it. This method is used in 'yield'.  */
  virtual int enqueue(Thread * _thr) {}; /* = NULL; */
  /* Puts given thread in ready queue. This method is used in 'resume'. */
public:
  Scheduler(); /* Instantiate a new scheduler.  This is done during OS startup. */
  /* -- START THE EXECUTION OF THREADS. */
  virtual int start();
  /* Start the execution of threads by yielding to first thread in ready queue.
     Has to be called AFTER at least one thread has been started (typically the idle thread). */
  /* -- SCHEDULING OPERATIONS */
  virtual int yield();
  /* Give up the CPU.  If another process is ready, make that process have the CPU. Returns 0 if ok. */
  int terminate_thread(Thread * _thr);
  /* Terminate given thread. The thread must be eliminated from any ready queue and its execution must be
     stopped. Special care must be taken if this is the currently executing thread. */
  int resume(Thread * _thr);
  /* Indicate that the process is ready to execute again.  The process is put on the ready queue.*/
};
```

Managing and Dispatching Threads (2)

```
class Scheduler {
private:
  int yield_to(Thread * new_thread);   /* Calls low-
protected:
  Thread * current_thread;
  /* -- MANAGEMENT OF THE READY QUEUE */
  virtual int remove_thread(Thread * _thr) {}; /* =
  /* Remove the Thread from any scheduler queues. */
  virtual Thread * first_ready() {}; /* = NULL;*/
  /* Removes first thread from ready queue and retur
  virtual int enqueue(Thread * _thr) {}; /* = NULL;
  /* Puts given thread in ready queue. This method i
public:
  Scheduler(); /* Instantiate a new scheduler.  This
  /* -- START THE EXECUTION OF THREADS. */
  virtual int start();
  /* Start the execution of threads by yielding to f
     Has to be called AFTER at least one thread has
  /* -- SCHEDULING OPERATIONS */
  virtual int yield();
  /* Give up the CPU.  If another process is ready,
  int terminate_thread(Thread * _thr);
  /* Terminate given thread. The thread must be eliminated from any ready queue and its execution must be
     stopped. Special care must be taken if this is the currently executing thread. */
  int resume(Thread * _thr);
  /* Indicate that the process is ready to execute again.  The process is put on the ready queue.*/
};
```

```
int Scheduler::yield() {
  int return_code = 0;

  /* -- GET NEXT THREAD FROM READY QUEUE. */
  Thread * new_thread = first_ready();

  if (!new_thread) {
    /* --- THERE IS NO OTHER THREAD READY       */
    /*     (THIS MUST BE THE IDLE THREAD, THEN) */
    return return_code;
  }
  else {
    /* --- GIVE CONTROL TO new_thread */
    return_code = yield_to(new_thread);

    /* THIS CODE IS EXECUTED AFTER A resume OPERATION. */
    return return_code;
  }
} /* of Scheduler::yield() */
```

Managing and Dispatching Threads (2)

```
class Scheduler {
private:
  int yield_to(Thread * new_thread);   /* Calls low-level dispatching mechanisms. */
protected:
  Thread * current_thread;
  /* -- MANAGEMENT OF THE READY QUEUE */
  virtual int remove_thread(Thread * _thr) {}; /* = NULL; */
  /* Remove the Thread from any scheduler queues. */
  virtual Thread * first_ready() {}; /* = NULL;*/
  /* Removes first thread from ready queue and returns it. This method is used in 'yield'.  */
  virtual int enqueue(Thread * _thr) {}; /* = NULL; */
  /* Puts given thread in ready queue. This method is used in 'resume'. */
public:
  Scheduler(); /* Instantiate a new scheduler.  This
  /* -- START THE EXECUTION OF THREADS. */
  virtual int start();
  /* Start the execution of threads by yielding to f
     Has to be called AFTER at least one thread has
  /* -- SCHEDULING OPERATIONS */
  virtual int yield();
  /* Give up the CPU.  If another process is ready,
  int terminate_thread(Thread * _thr);
  /* Terminate given thread. The thread must be elim
     stopped. Special care must be taken if this is the currently executing thread. */
  int resume(Thread * _thr);
  /* Indicate that the process is ready to execute again.  The process is put on the ready queue.*/
};
```

```
int Scheduler::resume(Thread * _thr) {

  /* This thread better not be on the ready queue. */
  assert(_thr->thread_state != THRD_READY);

  enqueue(_thr);

  return 0;

} /* Scheduler::resume() */
```

Managing and Dispatching Threads (2)

```
class Scheduler {
private:
  int yield_to(Thread * new_thread);   /* C
protected:
  Thread * current_thread;
  /* -- MANAGEMENT OF THE READY QUEUE */
  virtual int remove_thread(Thread * _thr)
  /* Remove the Thread from any scheduler q
  virtual Thread * first_ready() {}; /* = N
  /* Removes first thread from ready queue
  virtual int enqueue(Thread * _thr) {}; /
  /* Puts given thread in ready queue. This
public:
  Scheduler(); /* Instantiate a new schedul
  /* -- START THE EXECUTION OF THREADS. */
  virtual int start();
  /* Start the execution of threads by yiel
     Has to be called AFTER at least one th
  /* -- SCHEDULING OPERATIONS */
  virtual int yield();
  /* Give up the CPU.  If another process i
  int terminate_thread(Thread * _thr);
  /* Terminate given thread. The thread mus
     stopped. Special care must be taken if
  int resume(Thread * _thr);
  /* Indicate that the process is ready to
};
```

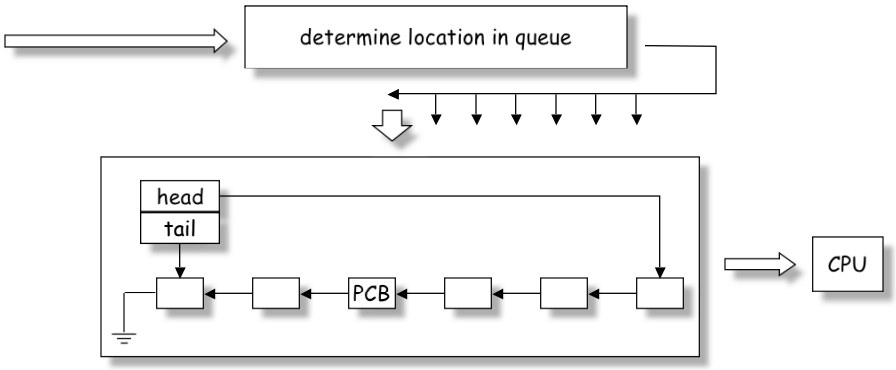
```
int Scheduler::terminate_thread(Thread * thr) {

  /* Call the scheduler-specific function to remove
     the Thread object from any queue.*/

  if (current_thread != thr) {
    if ((current_thread->thread_state == THRD_READY)
        || (current_thread->thread_state == THRD_INIT)) {
      remove_thread(thr);
    }
  }

  /* At this point the thread is not in any scheduler queue
     (anymore). The thread object is still around, though. */

  if (thr == current_thread) {
    /* The thread is committing suicide. We have to reschedule. */

    thr->thread_state = THRD_EXIT;

    /* This invokes the 'yield' method of the particular type of
       scheduler being used. The idea is that 'yield' will in turn
       call 'yield_to' to perform the dispatching. */
    yield();

    /* WE SHOULD NOT BE REACHING THIS PART OF THE CODE! */
    assert(FALSE);
  }
}
```

Managing and Dispatching Threads (2)

```
class Scheduler {
private:
  int yield_to(Thread * new_thread);   /* (
protected:
  Thread * current_thread;
  /* -- MANAGEMENT OF THE READY QUEUE */
  virtual int remove_thread(Thread * _thr)
  /* Remove the Thread from any scheduler q
  virtual Thread * first_ready() {}; /* = N
  /* Removes first thread from ready queue
  virtual int enqueue(Thread * _thr) {}; /*
  /* Puts given thread in ready queue. This
public:
  Scheduler(); /* Instantiate a new schedul
  /* -- START THE EXECUTION OF THREADS. */
  virtual int start();
  /* Start the execution of threads by yiel
     Has to be called AFTER at least one th
  /* -- SCHEDULING OPERATIONS */
  virtual int yield();
  /* Give up the CPU.  If another process i
  int terminate_thread(Thread * _thr);
  /* Terminate given thread. The thread mus
     stopped. Special care must be taken if
  int resume(Thread * _thr);
  /* Indicate that the process is ready to
};
```

```
int Scheduler::yield_to(Thread * new_thread) {

  int special_action = 0;
  int error_code     = 0;

  Thread * old_thread = current_thread;

  if (old_thread->thread_state == THRD_EXIT)
          special_action |= ACTION_EXIT;
  if (new_thread->thread_state == THRD_INIT)
          special_action |= ACTION_INIT;

  current_thread = new_thread;
  /* If everything goes well. */

  old_thread->thread_state = THRD_STOPPED;
  /* Have to do this here; will not have another chance
     later. */

  thread_yield(&(old_thread->thread_context),
               &(new_thread->thread_context),
               special_action);

  /* The following will never be reached if the thread
     was exiting. */

  return error_code;
}
```

Reminder: Structure of a Scheduler
(conceptual structure)



- Incoming process is put into right location in ready queue.
- Dispatcher always picks first element in ready queue.

Dispatching and Scheduling

```
class FIFOScheduler : public Scheduler {
protected:
  Queue    ready_queue;     /* The ready processes queue up here.          */

  virtual int remove_thread(Thread * thr) {
  /* Remove the Thread from the ready_queue. */
     int return_code = ready_queue.remove(thr);
     assert(return_code == 0);
     return return_code;
  }
  virtual Thread * first_ready() {
  /* Removes first thread from ready queue and returns it. This method is used in 'yield'.  */
     Thread * new_thread = (Thread*)ready_queue.get();
  }
  virtual int enqueue(Thread * _thr) {
  /* Puts given thread in ready queue. This method is used in 'resume'. */
     ready_queue.put(_thr);
  }

public:

  FIFOScheduler() : Scheduler(); ready_queue() {}
  /* Instantiate a new scheduler.  This has to be done during OS startup. */

};
```

Low-Level Dispatching, MIPS-style

```
LEAF(thread_yield)
    #  a0 :  pointer to current thread's context frame
    #  a1 :  pointer to new thread's context frame
    #  a2 .AND. ACTION_INIT != 0 -> new thread just initialized.
    #  a2 .AND. ACTION_EXIT != 0 -> old thread exits. do not save state.
    #     : other       -> simple context switch.

    li      t1, ACTION_EXIT
    and     t3, t1, a2
    bnez    t3, start_switch # -- IF THREAD EXISTS, SKIP STATE SAVING

    # IF THREAD IS EXITING, POINTER TO PROCESSOR STATE TABLE IS LIKELY INVALID.
    sw      s0, S0_OFF(a0)       # -- SAVE CURRENT STATE
    …
    sw      s6, S6_OFF(a0)
    sw      s7, S7_OFF(a0)
    sw      gp, GP_OFF(a0)
    sw      ra, RA_OFF(a0)
    sw      fp, FP_OFF(a0)
    sw      sp, SP_OFF(a0)
start_switch:
    lw      s0, S0_OFF(a1)   # -- LOAD REGISTERS FOR NEW TASK
    …
    lw      s7, S7_OFF(a1)
#   lw      gp, GP_OFF(a1)
    lw      ra, RA_OFF(a1)
    lw      fp, FP_OFF(a1)
    lw      sp, SP_OFF(a1)
```

(continue on next slide)

Low-Level Dispatching, MIPS-style (2)

```
(from previous slide:
    1. unless ACTION_EXIT, save state of old thread.
    2. load state of new thread.
}

    li      t1, ACTION_INIT
    and     t3, t1, a2
    beqz    t3, simple_switch

    # this is a new thread starting, load init PC and start from there.
    lw      t2, PC_OFF(a1)
    jalr    ra, t2

    # at this point the thread function has completed. stop the thread.
    # XXXXX NEED TO FILL IN CODE !!!!

simple_switch:
    # the new thread is all ready to go, just start.
    j       ra
END(thread_yield)
```

Simple Preemptive Scheduling

```
class RRScheduler : public FIFOScheduler {
private:
  unsigned int   time_quantum;
  Timer        * quantum_timer;
  friend class   EndOfQuantumEvent;

  void handle_end_of_quantum(EXCEPTION_CONTEXT * _xcp) {
     quantum_timer->set(time_quantum, _xcp->compare);
     if (task_ready()) {
             resume(current_thread);
             Scheduler::yield();
     }
  }
public:
  RRScheduler(unsigned int _quantum) : FIFOScheduler()
     time_quantum               = _quantum;
     EndofQuantumEvent * eoq_ev = new EndOfQuantumEvent(this);
     quantum_timer              = new Timer(eoq_ev);
  }
  virtual int start() {
     quantum_timer->set(time_quantum);
     FIFOScheduler::start();
  }
  virtual int yield() {
     quantum_timer->clear();
     quantum_timer->set(time_quantum);
     Scheduler::yield();
  }
};
```

```
class EndOfQuantumEvent : public TimerEvent {
private:
  RRScheduler * sched;
public:
  EndOfQuantumEvent(RRScheduler * _sched) {
    sched = _sched;
  }
  void event_handler(EXCEPTION_CONTEXT * _xcp) {
    clear_exl();
    sched->handle_end_of_quantum(_xcp);
  }
};
```