Operating System Concepts

Lecture 11a: Remote Procedure Call

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

What happens during a local procedure call?

- User program calls a library function, which **packs** arguments for the corresponding system call and raises the system call exception
- The system call handler **unpacks** the arguments and performs the requested operation
- The system call handler puts the result into a register and resumes the user process (or thread)
- Finally, the library function gets the system call's result and returns it to the caller (i.e. user program)

Remote Procedure Call (RPC)

- Calling a remote function should be as simple and as calling a local function
  - Example: Network File System (NFS)
- RPC server defines a list of functions it wishes to export
  - **stubs** (or wrappers) are used to simplify remote calls and hide details of communication
- A remote call is still much slower and less reliable than a local call
  - OS has to ensure that an RPC is acted upon exactly once
- See the map page of RPC: `rpc(3)`

RPC System

- Stub generator (or protocol compiler) takes the set of function calls that a server wishes to export (**procedure signatures**) and based on that it automatically packs function arguments and results into messages
  - benefit: avoiding mistakes
- A client-side stub handles the following operation
  - creates a message buffer
  - **packs** the function identifier and arguments into the buffer (aka message serialization or **marshalling** of arguments)
  - sends the message across the network to the destination RPC server (handled by the **run-time library**)
  - waits for the reply
  - **unpacks** the return code and/or results (aka message deserialization or **unmarshalling**)
  - returns to the caller
- Marshalling may require serializing objects, copying arguments passed by reference, etc.

RPC System

- Each message is addressed to an RPC daemon listening to a port on the RPC server
- A server-side stub
  - unpacks the message from the buffer to determine the called function and its parameters (**unmarshalling**)
  - calls the function with the provided parameters
  - packs the result into the buffer (**marshalling**)
  - sends the reply
- The RPC server is implemented as a concurrent server typically using a **thread pool**

RPC System

- The run-time library
  - locates the remote server using hostname (must be converted to IP address) and port number
  - converts byte ordering (endianness) if necessary
    - in big endian, bytes are stored from most significant to least significant; little endian is the opposite
  - sends data using UDP/IP or TCP/IP as a transport
    - so sockets will be used for communication
    - remote calls must be performed reliably and (at most) one time; so is TCP the right choice?
    - for performance reasons, many RPC systems are built on UDP and take care of timeout/retry and acknowledgment themselves

Remote Procedure Call (RPC)

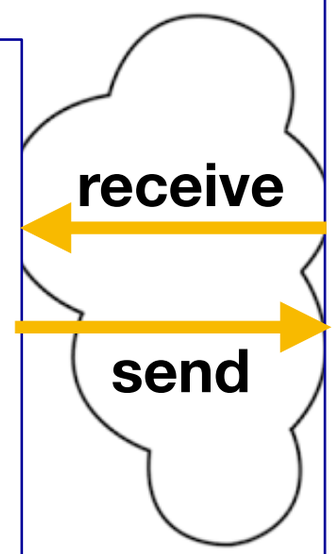
```
procedure P1
begin
    ...
    call Function(args)
    ...
end P1
```

call

return

```
procedure Function(args)
begin
  <marshall parameters>
  send(FunctionServer,params)
  receive(FunctionServer,results)
  <unpack results>
  return(results)
end Function
```

Client stub




```
procedure
realFunction(args)
begin
    ...
    return(results)
end realFunction
```

return

call

```
process FunctionServer
begin
  loop
    sender := select()
    receive(sender,params)
    <unpack parameters>
    call realFunction(args)
    <marshall results>
    send(sender,results)
  end loop
end FunctionServer
```

Server stub

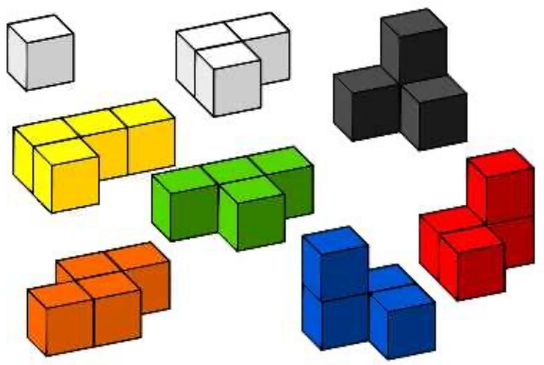
Operating System Concepts

Lecture 11b: CPU Scheduling

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

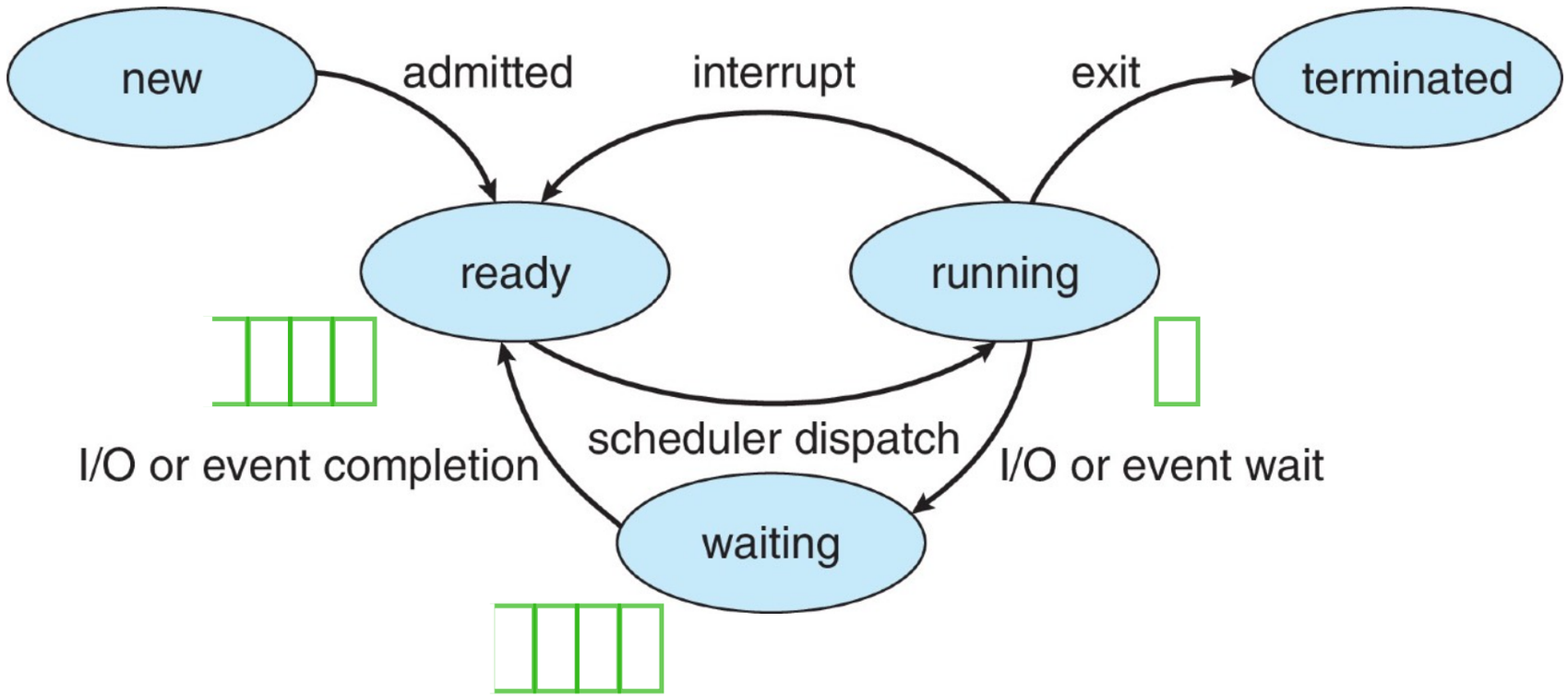
Today’s class

- Scheduling goals
  - switching CPU among processes can make computers more productive



Scheduling processes

Which process/thread to remove from ready queue?

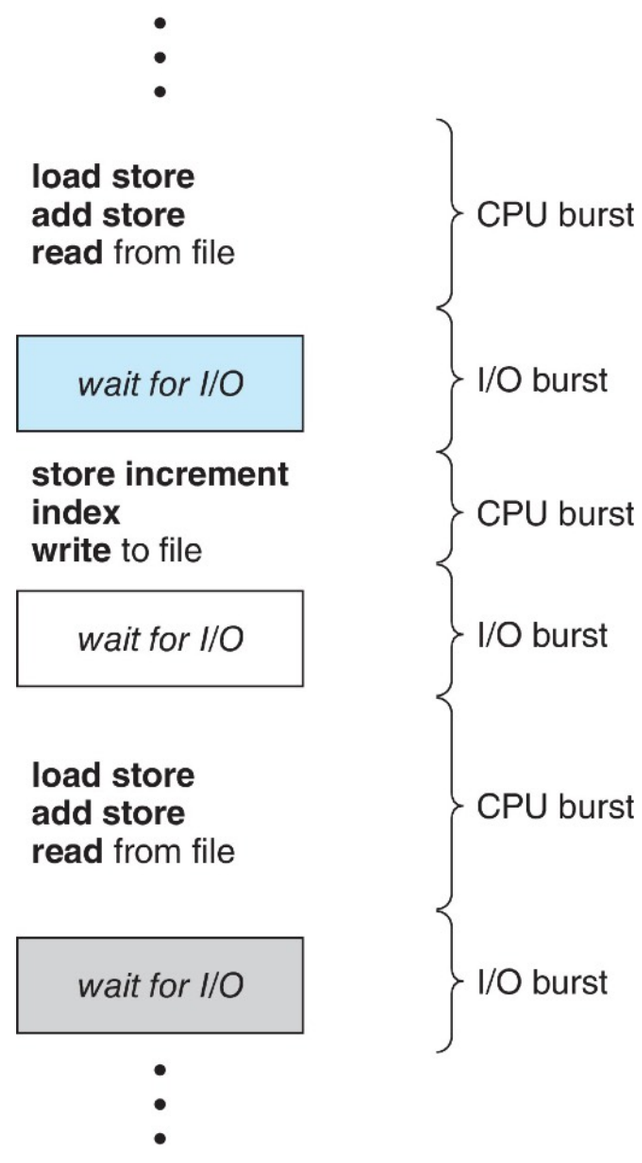


Scheduling is a difficult problem in general because of

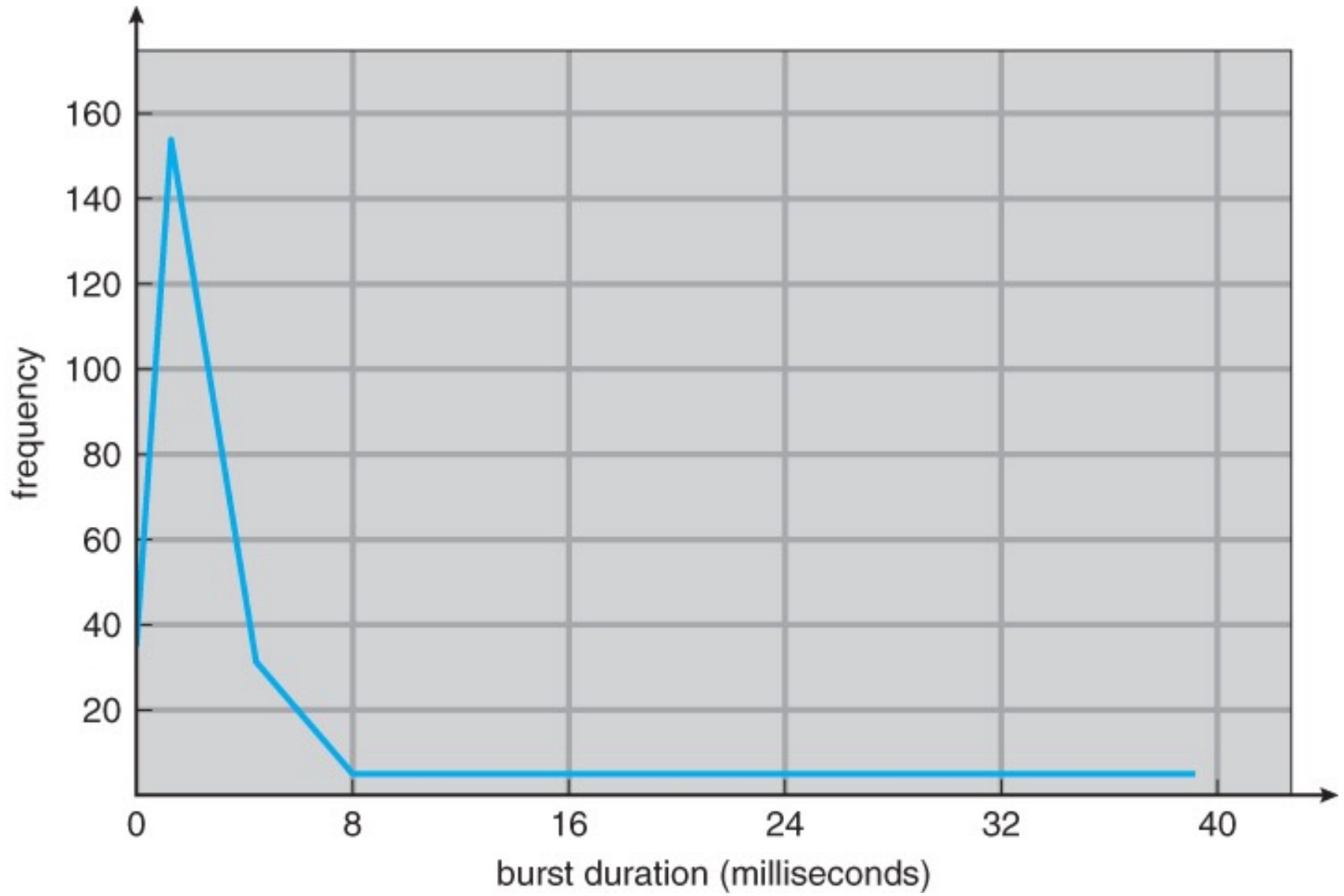
1. competing objectives (keeping users happy and the system fully utilized)
   - users care about getting their job done quickly
   - the system cares about overall efficiency, energy use, etc.
2. workload variability

Programs alternate between bursts of CPU and I/O activities

- Process execution consists of a cycle of CPU execution and I/O wait
  - CPU burst followed by I/O burst
  - CPU burst distribution is our main concern
- Three types of processes
  - CPU bound (more time spent doing computation)
  - I/O bound (more time spent doing I/O, CPU bursts are short)
  - balanced
- If all processes are I/O bound, the ready queue will almost always be empty, and if all of them are CPU bound then the I/O waiting queue will almost always be empty
  - best performance is achieved when there is a good mix of I/O and CPU bound processes



Histogram of CPU burst durations of a typical process



Definition

- Workload
  - a set of tasks for a system to run, each task has a specific arrival time and a burst length (and a **deadline** in real-time systems)
- Performance metrics
  - criteria considered for comparing scheduling policies (e.g., throughput, response time)
- Scheduling policy
  - decision about the order of executing tasks for a given workload
  - different performance achieved by different policies
- Overhead
  - extra work done by the scheduler (e.g., for context switching)
- Fairness
  - fraction of resources (e.g. CPU cycles) provided to each task
  - **max-min fairness**: maximize the minimum allocation given to a task

Scheduling

- Long-term scheduling (job scheduling)
  - how does the OS determine the degree of multiprogramming, i.e., the number of jobs executing at once in the primary memory?
  - admitting jobs and loading them into memory for execution to provide a balanced mix of jobs
- Short-term scheduling (CPU scheduling)
  - how does the OS select a process from the ready queue to execute?

Scheduling

- Off-line scheduling computes a schedule given the entire set of tasks (assuming the knowledge of arrival times and burst lengths)
- On-line scheduling makes decisions as tasks arrive
  - which one yields better performance?

Scheduling opportunities

- The kernel runs the scheduler at least when
  - a process switches from running to waiting
  - an interrupt occurs (e.g., timer expires)
  - a process is created or terminated
- Non-preemptive system
  - the scheduler must wait for one of these events
- Preemptive system
  - the scheduler can interrupt a running process

Performance criteria for scheduling

- **CPU Utilization** (higher is better)
  - percentage of time CPU is busy
- **Throughput** (higher is better)
  - number of processes completing in a unit of time (e.g., in one second)
  - overhead reduces the throughput
- **Waiting time** (lower is better)
  - total amount of time a process spends in the ready queue
- **Turnaround time** (lower is better)
  - length of time it takes to run process from initialization to termination, including waiting time
- **Response time** (lower is better)
  - what user sees from keypress to character on screen
  - time between when process enters the ready queue and it finishes execution on CPU (first response is produced)

How to choose a scheduling policy?

- Choose a CPU scheduler that optimizes all criteria simultaneously
  - maximize CPU utilization and throughput; minimize turnaround time, waiting time, and response time (**this is not usually possible**)
- Choose a scheduling algorithm that implements a particular policy
  - minimize average response time - provide output to user as quickly as possible and process their input as soon as it is received
  - minimize variance of response time - in interactive systems, predictability may be more important than a low average with a high variance
  - maximize throughput - has two components:
    - minimize overhead (OS overhead, context switching)
    - efficient use of system resources (CPU, I/O devices)
  - minimize waiting time - give each process the same amount of time on the processor (is it fair?)
    - this might actually increase the average response time