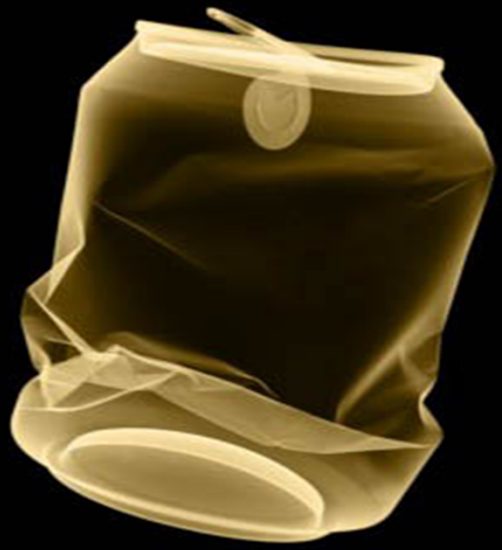
Process

Instructor: Hengming Zou, Ph.D.

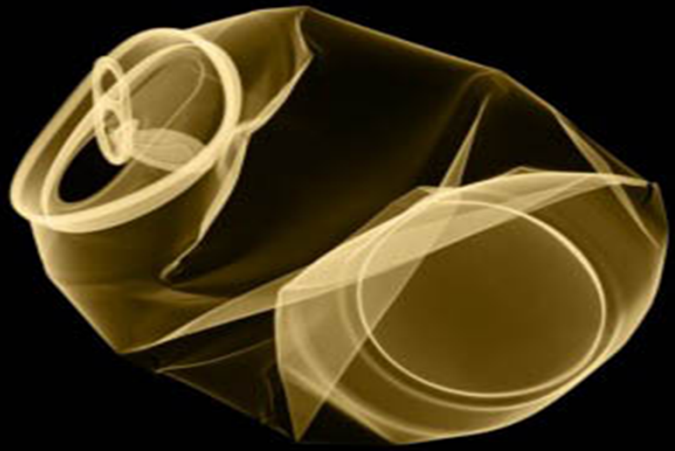
In Pursuit of Absolute Simplicity 求于至简，归于永恒

Process

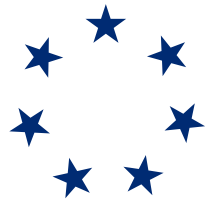


IPC

Scheduling



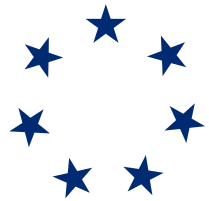
Content



- Process Basics:
  - Processes, process modeling, process state, process space
  - Benefit of multiprogramming
- IPC-Inter Process Communication
  - Traditional UNIX IPC (pipe, named pipe, socket, signal)
  - Shared memory, message queue, semaphore
- CPU Scheduling
  - Scheduling Objective
  - FCFS, Round Robin, STCF, Real-Time Scheduling

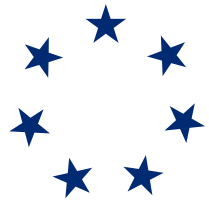
Process Basics

Definition of A Process



- Informal
  - A program in execution
  - A running piece of code along with
    - →all the things the program can read/write
- Formal
  - One or more threads in their own address space
- Note that process != program

The Need for Process

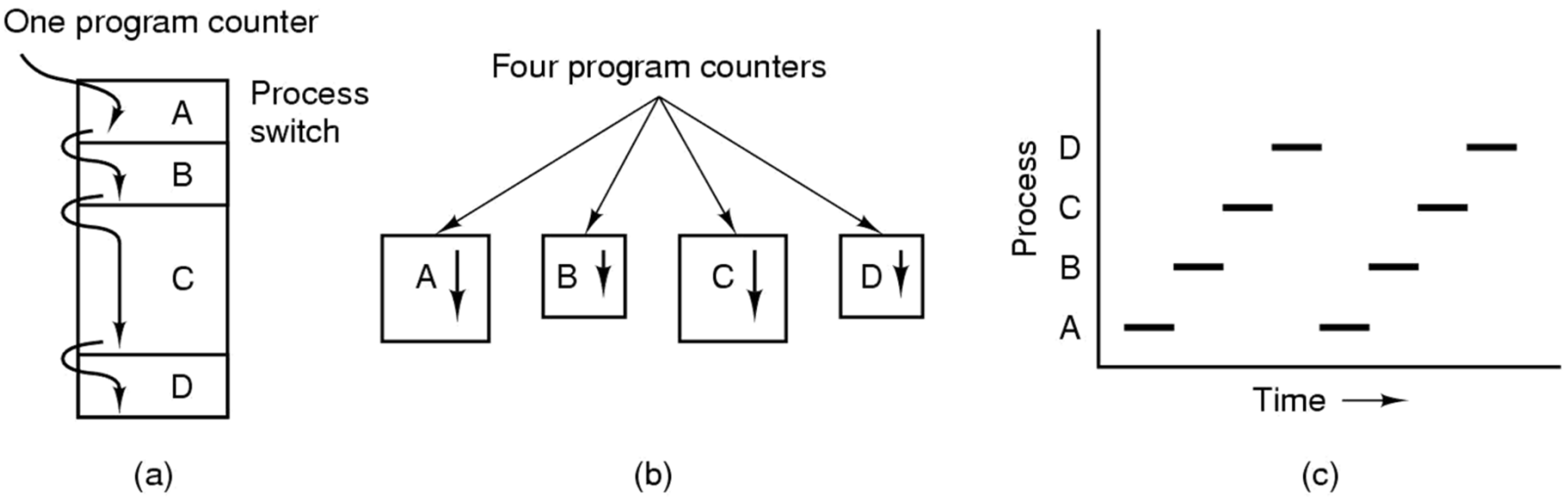


- What is the principle motivation for inventing process?
  - To support multiprogramming

The Process Model

- Conceptual viewing of the processes
- Concurrency
  - Multiple processes seem to run concurrently
  - But in reality only one active at any instant
- Progress
  - Every process makes progress

Multiprogramming of 4 Programs

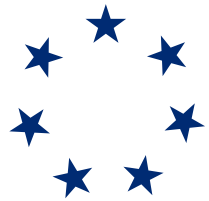


Programs in memory

Conceptual view

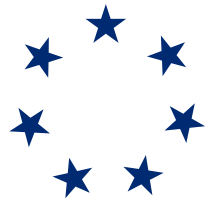
Time line view

Process Creation and Termination



- Principal events that cause process creation
  - System initialization
  - Execution of a process creation system
  - User request to create a new process
- Conditions which terminate processes
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal error (involuntary)
  - Killed by another process (involuntary)

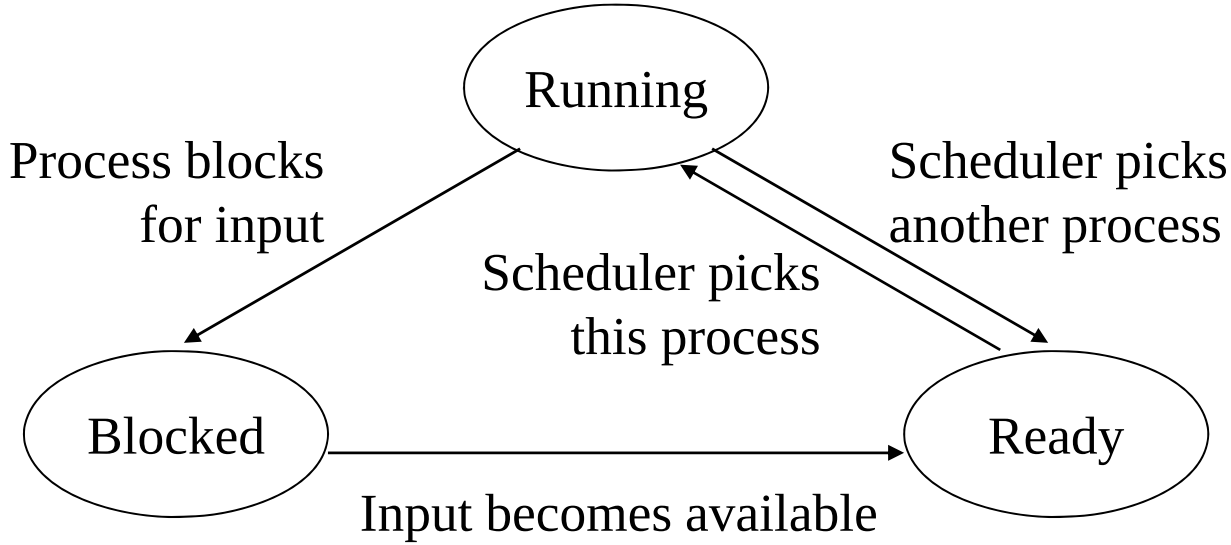
Process Hierarchies



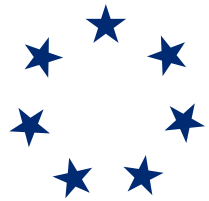
- Parent creates a child process
- Child processes can create its own process
- Processes creation forms a hierarchy
  - UNIX calls this a "process group"
  - Windows has no concept of such hierarchy,
    - →i.e. all processes are created equal

Process States

- Possible process states
  - running, blocked, ready

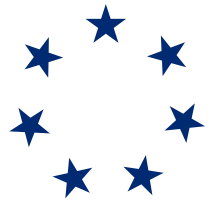


Process Space



- Also called Address Space
- All the data the process uses as it runs
- Is the unit of state partitioning
  - Each process occupies a different state of the computer
- Passive (acted upon by the process)
- Play analogy:
  - all the objects on the stage in a play
- Main topic:
  - How multiple processes spaces can share a single physical memory efficiently and safely

Process Management

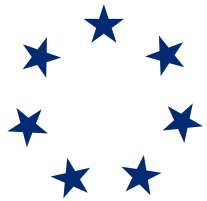


- Who manages process and space?
  - The operating systems
- How does OS achieve it?
  - By maintain information about processes
  - i.e. use process tables

Fields of A Process Table

- Registers, Program counter, Status word
- Stack pointer, Priority, Process ID
- Parent group, Process group, Signals
- Time when process started
- CPU time used, Children's CPU time, etc.

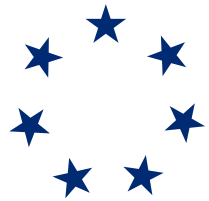
Process Creation



- Process are created and destroyed all the time
- Many steps involved in Process Creation

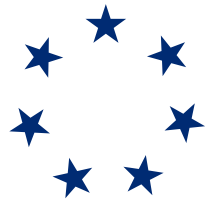
- Steps in Process Creation
  - Allocate process control block
  - Read code from disk and store into memory
  - Initialize machine registers
  - Initialize translator data (page table and PTBR)
  - Set processor mode bit to “user”
  - Jump to start of program

Steps in Process Creation



- Need hardware support for last few steps
- Otherwise processor executing in user mode can't
  - access the kernel's jump instruction
- Switching from kernel to user process is the same as last 4 steps above
  - e.g. after a system call completes

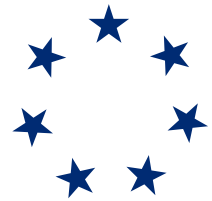
Examples of Process Creation



- Unix separates process creation into two steps
- Unix fork:
  - create a new process (with 1 process)
  - Address space of new process is a copy of parent's
- Unix exec:
  - overlay the new process's address space with the specified program
  - and jump to its starting PC

    →this loads the new program

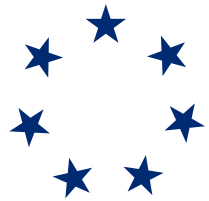
Examples of Process Creation



- Example:
- Parent process wants to fork a child to do a task.
- Any problem with having the new process be an exact copy of the parent?

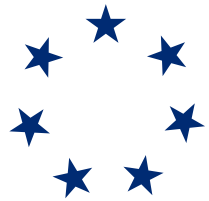
- Why does Unix fork copy the parent's entire address space,
  - just to throw it out and start with the new address space?
- Unix provides the semantic of copying the parent's entire address space
  - but does not physically copy the data until needed

Examples of Process Creation



- Separating fork and exec gives maximum flexibility
  - for the parent process to pass information to the child
- Common special case:
  - fork a new process that runs the same code as parent

Alternative Process Creation

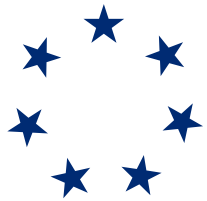


- Windows creates processes with a single call
  - CreateProcess
- Unix's approach gives the flexibility of
  - sharing arbitrary data with child process
- Window's approach allows
  - the program to share the most common data via parameters

Benefit of Multiprogramming

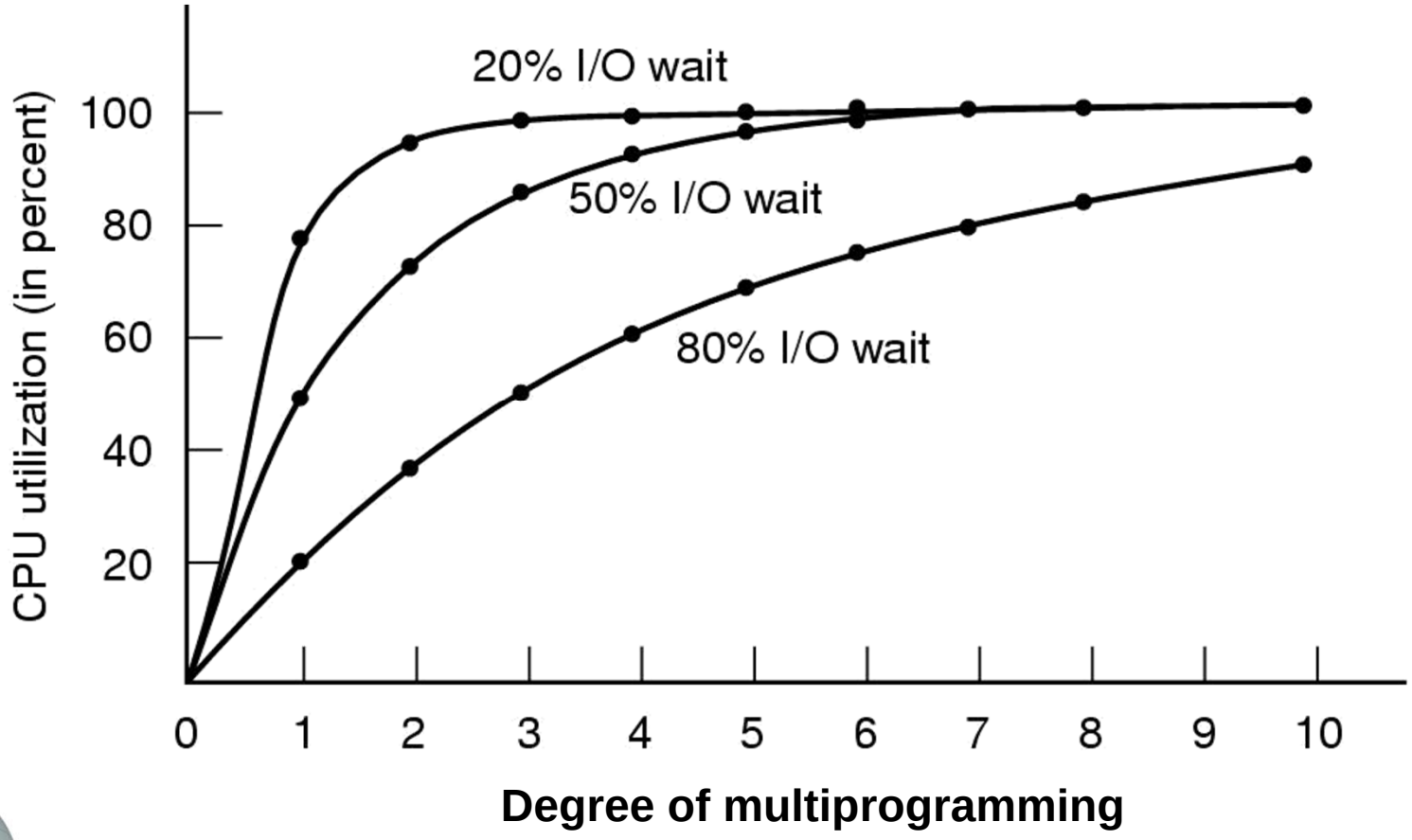
- Improve resource utilization
  - All resource can be kept busy with proper care
- Improve response time
  - Don’t need to wait for previous process to finish to receive a feedback

CPU Utilization of Multiprogramming

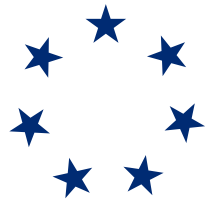


- Assume processes spend 80% time wait for I/O
- CPU utilization for uni-programming: 20%
- CPU utilization for two processes: 36%
  - Rough estimation only (i.e. ignore overhead)
  - CPU utilization = $1 - 0.8 \times 0.8 = 0.36$
- CPU utilization for three processes: 48.8%

CPU Utilization of Multiprogramming



Multiprogramming System Performance



- Given:
  - Arrival and work requirements of 4 jobs
  - CPU utilization for 1 – 4 jobs with 80% I/O wait
- Plot:
  - Sequence of events as jobs arrive and finish
  - Show amount of CPU time jobs get in each interval

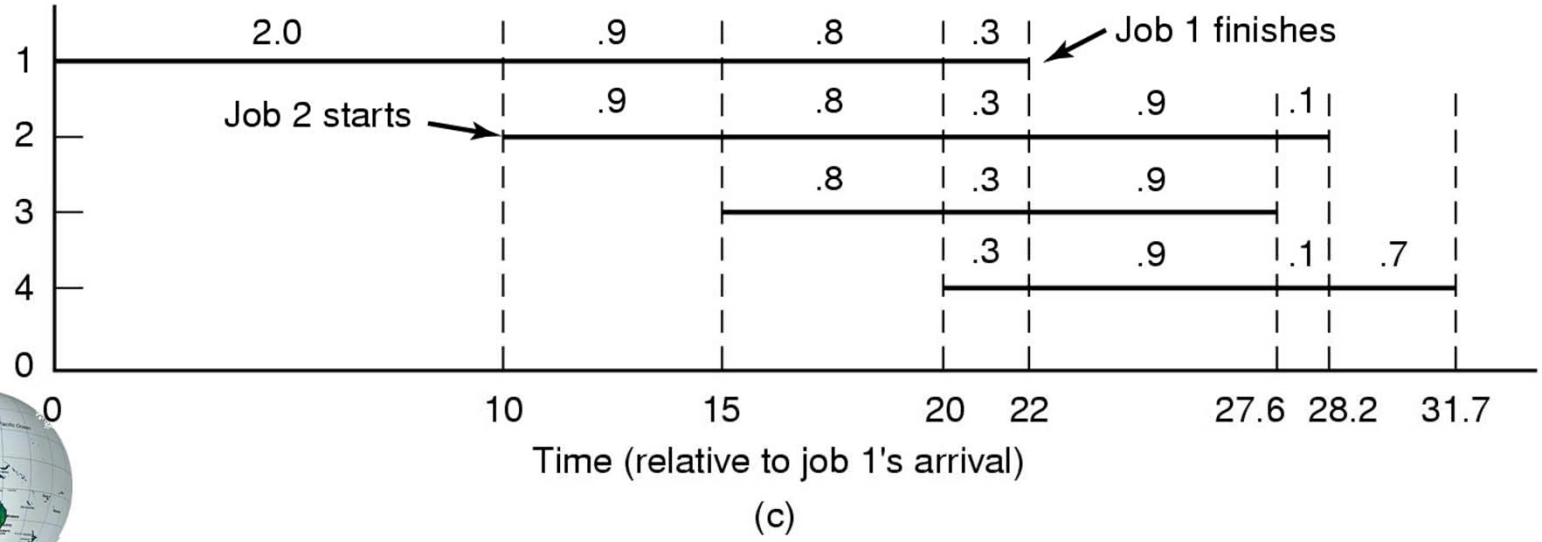
Multiprogramming System Performance

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

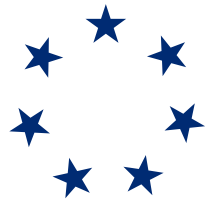
	# Processes 1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

Multi-process Issues

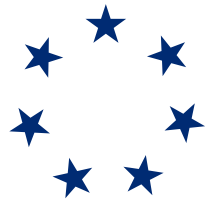


- How to allocate physical memory between processes?
  - Virtual memory deals with it.
- Resource allocation is an issue
  - whenever sharing a single resource among multiple users

    →e.g. CPU scheduling
- Often a tradeoff between globally optimal and fairness

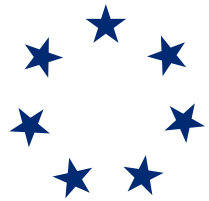
Inter-Process Communication

Inter Process Communication

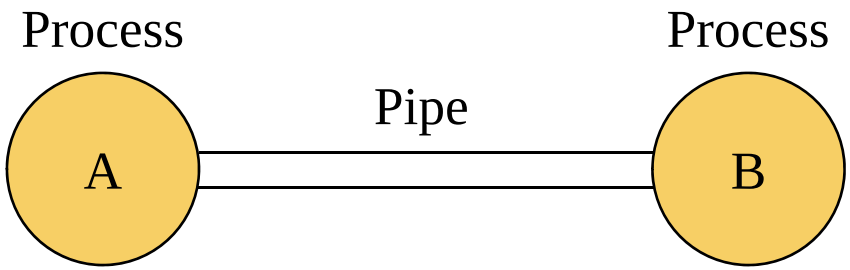


- Traditional IPC mechanism
  - Pipe, named pipe, socket,
  - Signal (already covered in monitor)
- Higher level IPC
  - Semaphore (already covered in synchronization)
  - Shared memory, LPC, RPC

Pipe

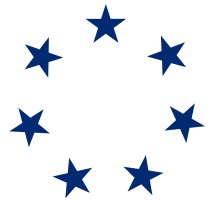


- A pipe is used for communication between related processes

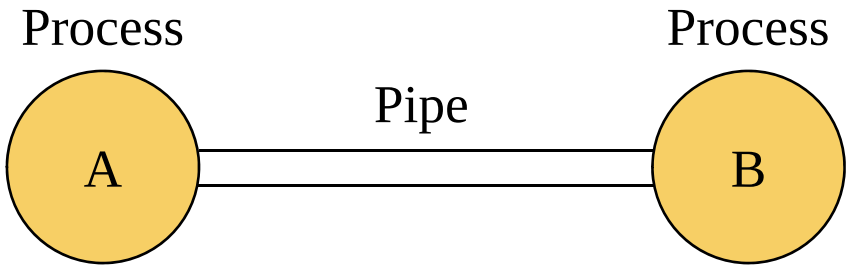


- A pipe is similar to a file (but not a file)
  - one process write to one end of the file
  - another process read from the other end of the file
- Pipe I/O is similar to file I/O
- A process create a pipe by calling the pipe system call

Named Pipe

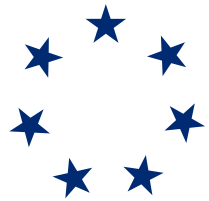


- A named pipe is used for communication between un-related processes



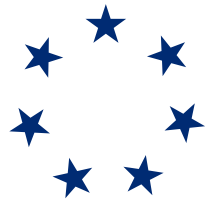
- A named pipe is FIFO queue
- It shares the name space with the file system
  - Thus it is easy to communicate between un-related processes
- A process create a FIFO pipe by calling the programming API mkfifo

Other Traditional IPC



- Socket:
  - A communication abstraction similar to a file
  - It has two ends with each end binds to a process
  - It can be local or remote
  - A process create a socket by calling socket creation APIs
- Signal:
  - A kernel object sent by one process to another process
  - Signal can be caught or ignored

Higher Level IPC

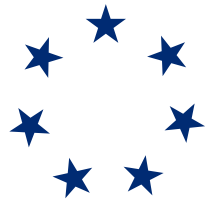


- Semaphore
  - Extension of signal by accumulating signals that have not been processed
  - Semaphore is more a synchronization mechanism

- Shared Memory:
  - A process can create a shared segment of memory
  - Other processes can attach to the shared memory segment
  - Any change to the shared memory is visible to all attached processes
- Message Queue
  - Through message passing

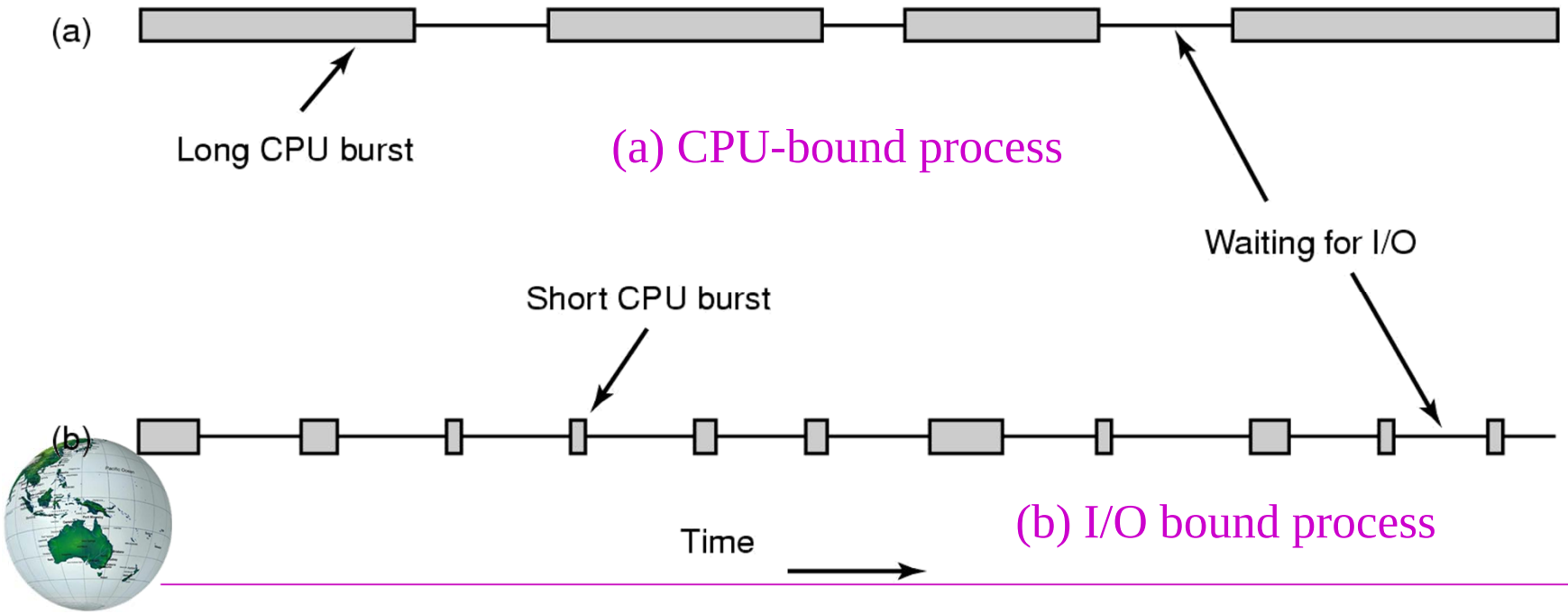
Process/CPU Scheduling

CPU Scheduling

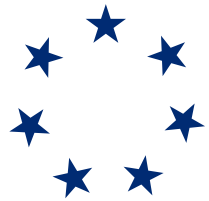


- How should one choose next process to run?
- What are the goals of the CPU scheduler?

- Bursts of CPU usage alternate with periods of I/O wait

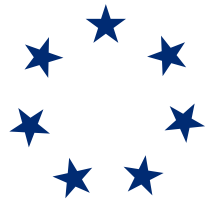


CPU Scheduling Objective



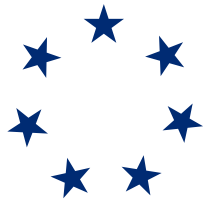
- Minimize average response time
  - average elapsed time to do each job
- Maximize throughput of entire system
  - rate at which jobs complete in the system
- Fairness
  - share CPU among processes in some “equitable” manner

Scheduling Algorithm Goals



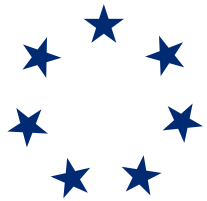
- All systems
- Fairness:
  - giving each process a fair share of the CPU
- Policy enforcement:
  - seeing that stated policy is carried out
- Balance:
  - keeping all parts of the system busy

Scheduling Algorithm Goals



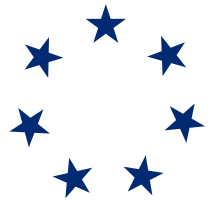
- Batch systems
- Throughput:
  - maximize jobs per hour
- Turnaround time:
  - minimize time between submission and termination
- CPU utilization:
  - keep the CPU busy all the time

Scheduling Algorithm Goals



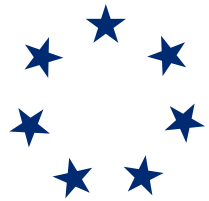
- Interactive systems
  - Response time – respond to requests quickly
  - Proportionality – meet users' expectations
- Real-time systems
  - Meeting deadlines – avoid losing data
  - Predictability – avoid quality degradation in multimedia systems

FCFS



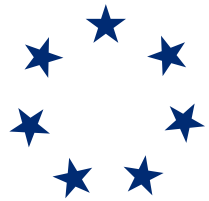
- First-Come, First-Served
- FIFO ordering between jobs
- No preemption (run until done)
  - process runs until it calls yield() or blocks on I/O
  - no timer interrupts

FCFS



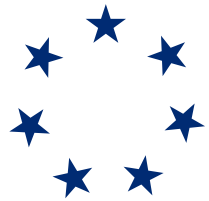
- Pros and cons
- + simple
- - short jobs get stuck behind long jobs
- - what about the user’s interactive experience?

FCFS

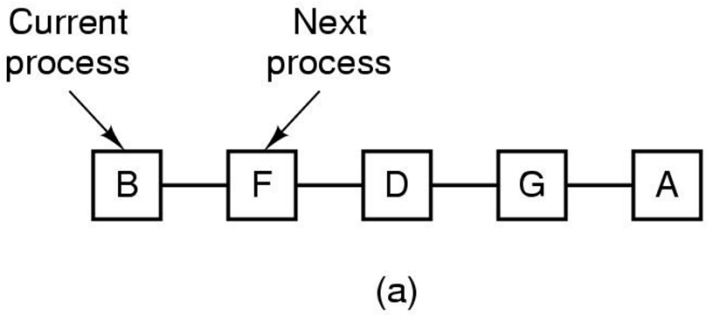


- Example
  - job A takes 100 seconds
  - job B takes 1 second
  - time 0: job A arrives and starts
  - time 0+: job B arrives
  - time 100: job A ends (response time=100); job B starts
  - time 101: job B ends (response time = 101)
- average response time = 100.5

Round Robin



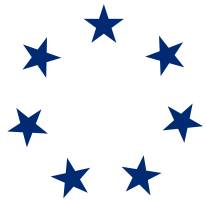
- Goal:
  - improve average response time for short jobs
- Solution:
  - periodically preempt all jobs (viz. long-running ones)



list of run-able processes

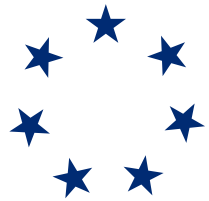
list of run-able processes
after B uses up its quantum

Round Robin



- Is FCFS or round robin more “fair”?
- Example
  - job A takes 100 seconds, job B takes 1 second
  - time slice of 1 second: a job is preempted after running for 1 second
  - time 0: job A arrives and starts
  - time 0+: job B arrives
  - time 1: job A is preempted; job B starts
  - time 2: job B ends (response time = 2)
  - time 101: job A ends (response time = 101)
- average response time = 51.5

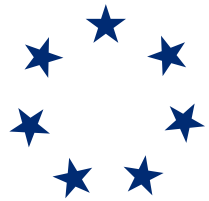
Round Robin



- Does round-robin always achieve lower response time than FCFS?

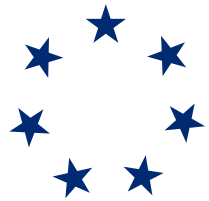
- Pros and cons
- + good for interactive computing
- - round robin has more overhead due to context switches

Round Robin



- How to choose time slice?
- big time slice:
  - degrades to FCFS
- small time slice:
  - each context switch wastes some time
- typically a compromise
  - 10 milliseconds (ms)
- if context switch takes .1 ms
  - then round robin with 10 ms slice wastes 1% of CPU

STCF

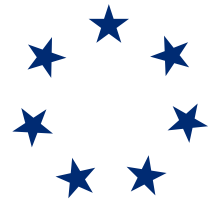


- Shortest Time to Completion First
- STCF: run whatever job has the least amount of work to do before it finishes or blocks for an I/O

- STCF-P: preemptive version of STCF
- if a new job arrives that has less work than the current job has remaining
  - then preempt the current job in favor of the new one

- Idea is to finish the short jobs first
  - Improves response time of shorter jobs by a lot

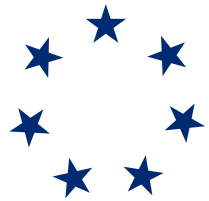
STCF



- STCF gives optimal response time among non-preemptive policies
- STCF-P gives optimal response time among preemptive policies (and non-preemptive policies)

- Is the following job a “short” or “long” job?
  - while(1) {
  - use CPU for 1 ms
  - use I/O for 10 ms
  - }

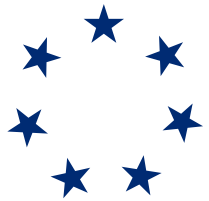
STCF



- Pros and cons
- + optimal average response time
- - unfair
  - Short jobs can prevent long jobs from ever getting any CPU time
  - (i.e. starvation)
- - needs knowledge of future

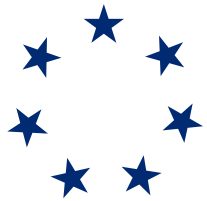
- STCF and STCF-P need knowledge of future
  - it’s often very handy to know the future :-)
  - how to find out the future time required by a job?

Example



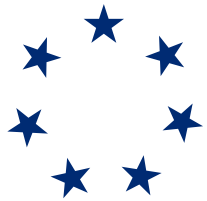
- job A
- compute for 1000 ms
- job B
- compute for 1000 ms
- job C
- while(1) {
- use CPU for 1 ms
- use I/O for 10 ms
- }

Example



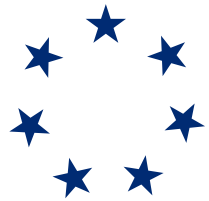
- C can use 91% of the disk by itself
- A or B can each use 100% of the CPU
- What happens when we run them together?
- Goal: keep both CPU and disk busy

Example



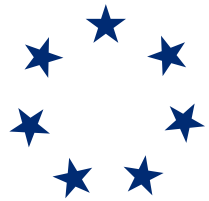
- FCFS:
  - if A or B run before C,
  - they prevent C from issuing its disk I/O for
    - →up to 2000 ms
- Round Robin with 100 ms time slice
  - CA---------B---------CA---------B---------...
  - Disk is idle most of the time that A & B are running
    - →about 10 ms disk time every 200 ms

Round Robin with 1ms



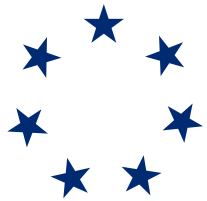
- with 1 ms time slice
  - CABABABABABCABABABABABC...
- C runs more often,
  - so it can issue its disk I/O almost as soon as its last disk I/O is done
- Disk is utilized almost 90% of the time
- Little effect on A or B's performance
- General principle:
  - first start the things that can run in parallel
- problem:
  - lots of context switches (and context switch overhead)

STCF-P



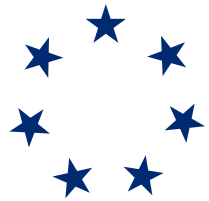
- Runs C as soon as its disk I/O is done
  - because it has the shortest next CPU burst
  - CA-------CA-------------CA--------- ...

Other Scheduling

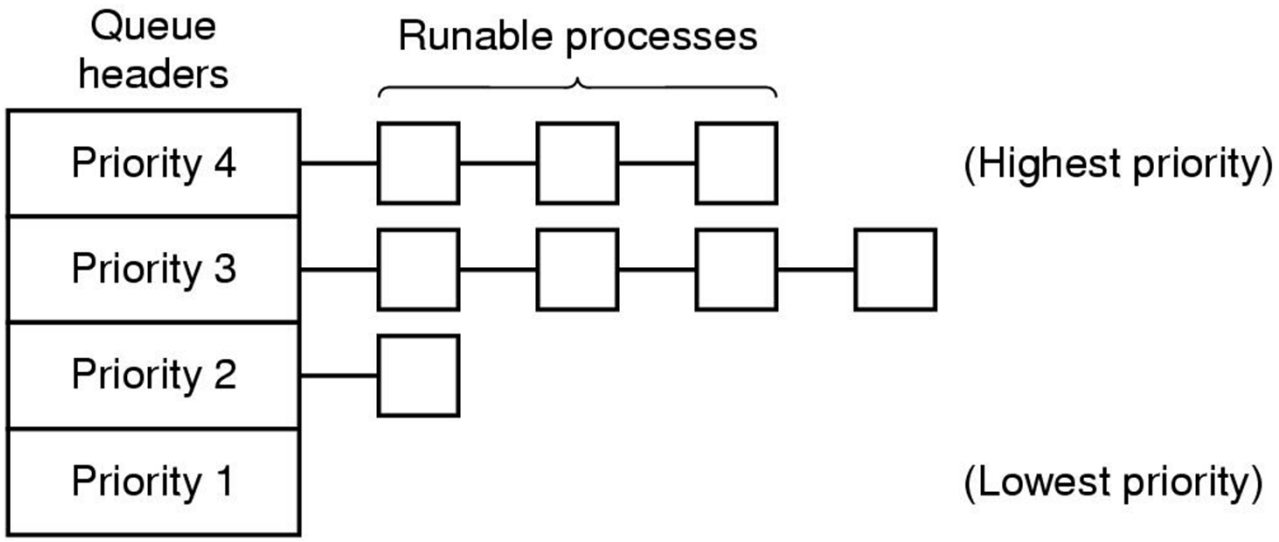


- Priority Scheduling
  - Assign each process a priority
  - Run highest priority process until it blocks or exits
  - Alternative, decrease priority at each clock tick
- Guaranteed scheduling
  - Each process runs 1/n fraction of time
- Lottery scheduling
  - Issue lottery tickets to process & schedule accordingly
- Fair share scheduling per user

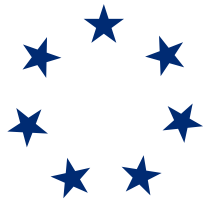
Other Scheduling



- Hybrid scheduling (Multiple queue)
  - Dividing priority into classes
  - Dynamically adjust a process's priority class

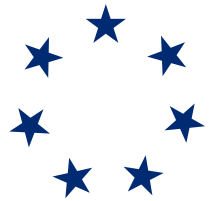


Priority Inversion



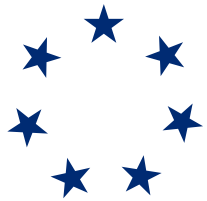
- A scenario where a low priority task holds a shared resource
  - that is required by a high priority task
- high priority task blocked until low priority task has released resource
  - effectively "inverting" the relative priorities of the two tasks.
- If some medium priority task, one that does not depend on the shared resource, attempts to run in the interim, it will take precedence over both
- May lead to system malfunction or the triggering of pre-defined actions.
  - a watchdog timer would go off and initiate a total system reset.
  - Mars Pathfinder is a classic example of problems
- Also reduce the perceived performance of the system.

Priority Inversion Solution



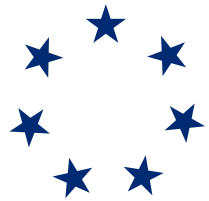
- Disabling all interrupts
  - to protect critical sections
- A priority ceiling
  - shared mutex process (that runs the operating system code) has a characteristic (high) priority of its own
- Priority inheritance
  - whenever a high priority task has to wait for some resource shared with an executing low priority task, the low priority task is assigned the priority of the highest waiting priority task for the duration

Real-Time Scheduling



- So far, we've focused on average-case analysis
  - average response time, throughput
- Sometimes, the right goal is to get each job done before its deadline
  - irrelevant how much before deadline job completes

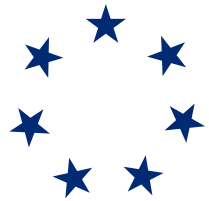
Real-Time Scheduling



- Video or audio output.
  - E.g. NTSC outputs 1 TV frame every 33 ms
- Control of physical systems
  - e.g. auto assembly, nuclear power plants

- This requires worst-case analysis
- How do we do this in real life?

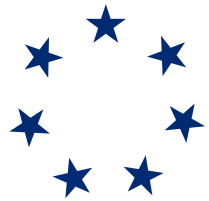
EDF-Earliest-Deadline First



- Preemptive dynamic scheduling algorithm
- Always run the job that has the earliest deadline
  - i.e. the deadline coming up next
- If a new job arrives with an earlier deadline
  - preempt the running job and start the new one
- **A set of tasks is schedulable if** $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$

EDF is optimal
it will meet all deadlines if it's possible to do so

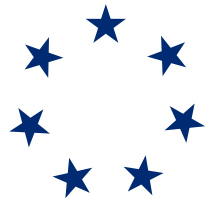
EDF



- Example
- job A: takes 15 seconds, deadline is 20 seconds after entering system
- job B: takes 10 seconds, deadline is 30 seconds after entering system
- job C: takes 5 seconds, deadline is 10 seconds after entering system

- Suppose A, B, C arrive in the system at almost the same time
- time--->
  - 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85
  - C A B

Rate-Monotonic Scheduling



- Non-preemptive static scheduling algorithm
- Schedule process according to its execution time/period rate
- Given
  - *m* periodic events
  - event *i* occurs within period P_i and requires C_i seconds
- Then the load can only be handled if

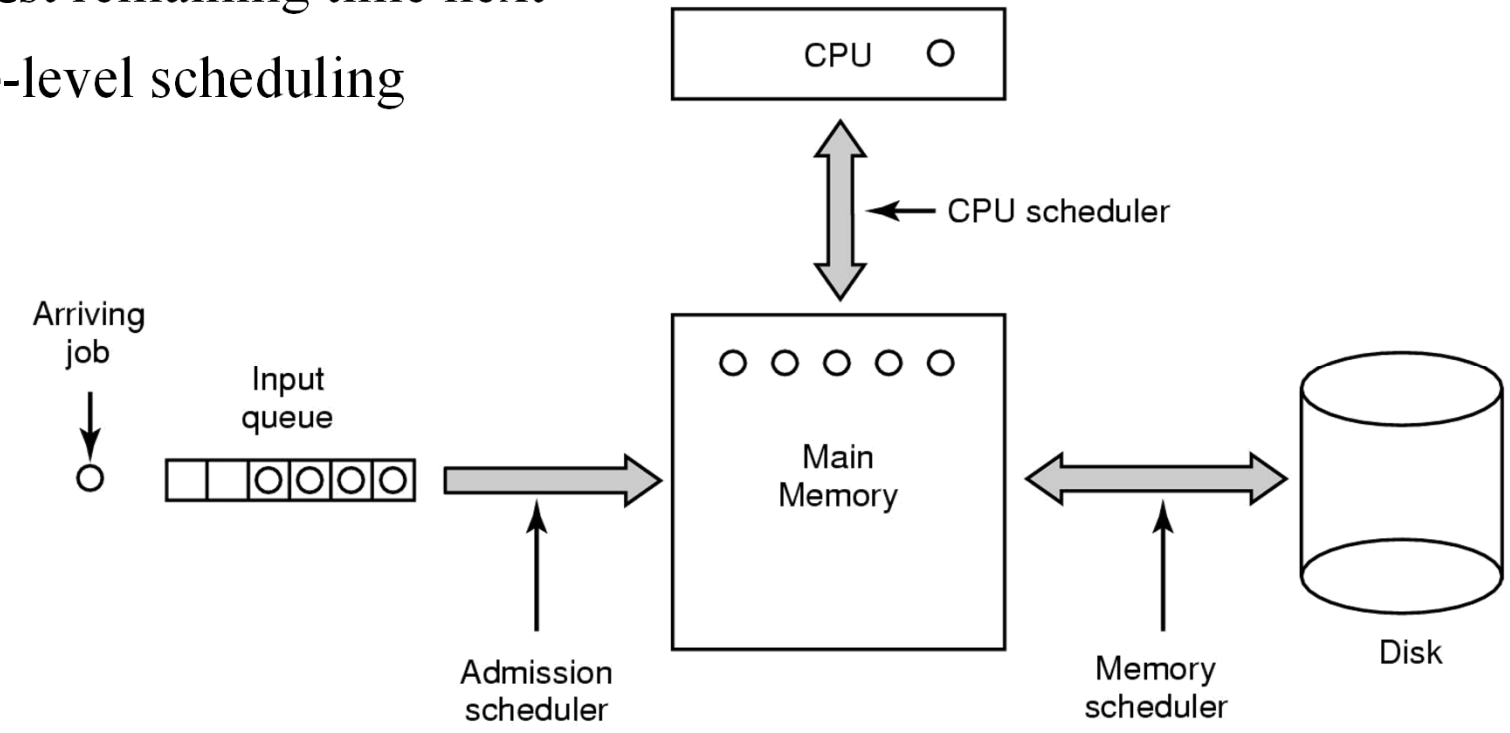
$$U = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(\sqrt[n]{2} - 1)$$

RMS is optimal in all static priority scheduling

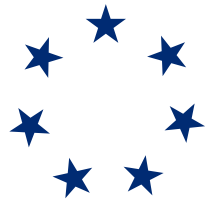
What is the implication to us?

Scheduling in Batch Systems

- First come first serve
- Shortest job first
- Shortest remaining time next
- Three-level scheduling



Policy versus Mechanism



- Separate what is allowed to be done
  - with how it is done
- Important in process scheduling
  - a process knows which of its children processes are important
  - and need priority
- Scheduling algorithm parameterized
  - mechanism in the kernel
- Parameters filled in by user processes
  - policy set by user process


Thoughts Change Reality
意念改变现实