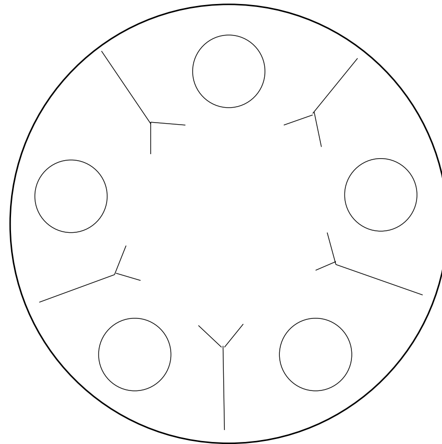
Solutions for IPC problems (Cont.)

- Message passing (Cont.)
  - Design issues
    - Blocking vs. nonblocking send/receive
    - Message format
    - Queueing discipline
  - Solving producer consumer problem with message passing
- Equivalence of IPC primitives
  - Semaphores ⇒ monitors
  - Semaphores ⇒ message passing
  - Monitors ⇒ semaphores

Some Classical IPC problems

The dining philosophers problem

- **Problem description**



  - **Five philosopher sit around a round table, and each of them has one fork.**
  - **Activities: eating and thinking.**
  - **To eat, need two adjacent forks.**
  - **Goal: no starvation.**

- **Useful for modeling processes that are competing for exclusive access to a limited number of resources, such as tape drive or other I/O devices.**

- **Problems with solution 2:**

  Assume Philosophers 1 and 4 eat for a long time and Philosophers 2 and 3 eat for a short time. A possible execution sequence:

  1. **4 and 1 eating.**
  2. **0 and 3 become hungry (blocked).**
  3. **4 finishes eating and checks neighbors.**
  4. **0 IS NOT ALLOWED TO EAT, but 3 is allowed to eat.**
  5. **4 becomes hungry again.**
  6. **3 finishes eating and allows 4 to eat again.**
  7. **1 finishes, 0 IS STILL NOT ALLOWED TO EAT because of 4.**
  8. **2 is allowed to eat.**
  9. **1 becomes hungry again.**
  10. **2 finishes and allows 1 to eat.**
  11. **repeat, 0 IS NEVER ALLOWED TO EAT.**

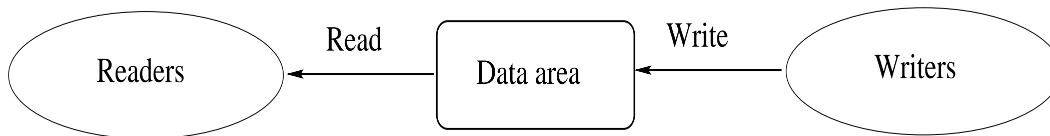
- **A working solution**

```
#define N 5
typedef int semaphore;
semaphore fork[N];   /* initially 1; */
semaphore room = 4;
/* allow 4 in the dining room */

philosopher(i)
int i;
{while (TRUE) {
think();
down(room);   /* get into dining room */
down(fork[i]);   /* get left fork */
down(fork[(i+1)%5)]);   /* get right fork */
eat();
up(fork[(i+1)%5)]); /*put back right fork*/
up(fork[i]);   /* put back left fork */
up(room);   /* get out of dining room */
}
}
```

The readers and writers problem

- **Problem description:**



  - **A data area (file or memory) shared among a number of processes.**
  - **Some processes (readers) only read the data area.**
  - **Other processes (writers) only write to the data area.**

- **Conditions must be satisfied:**

  1. **Any number of readers may simultaneously read the data area.**
  2. **Only one writer at a time may write to the data area.**
  3. **If a writer is writing to the data area, no readers may read it.**

- **Special type of mutual exclusion problem. Special solution can do better than general solution.**

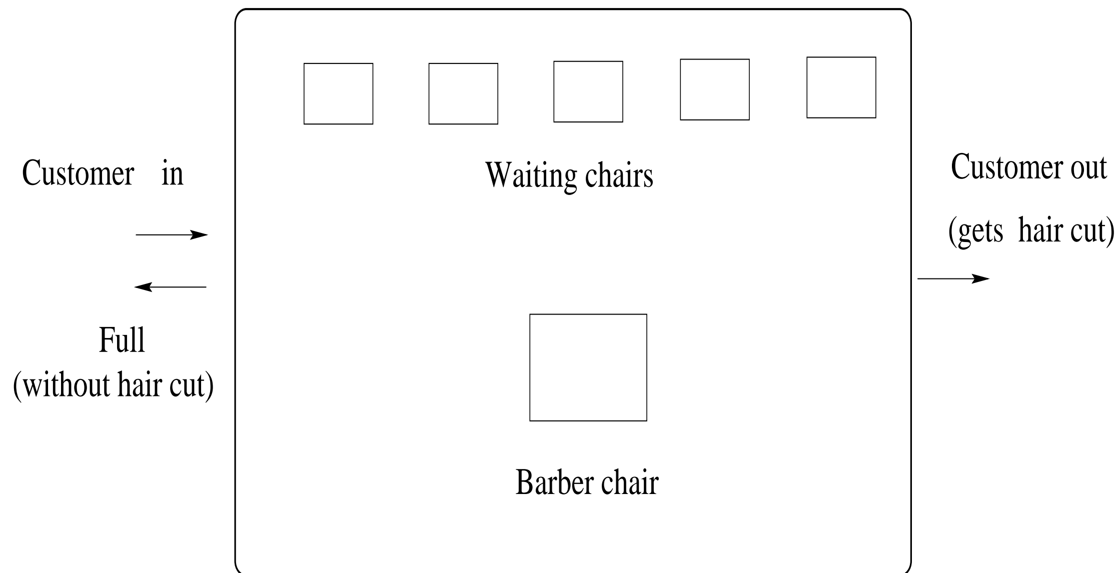
- **Solution one:**
  Readers have priority. Unless a writer is currently writing, readers can always read the data.

- **Solution two:**
  Writers have priority.
  Guarantee no new readers are allowed when a writer wants to write.

- **Other possible solutions:**
  Weak reader's priority or weak writer's priority.
  Weak reader's priority: An arriving reader still has priority over waiting writers. However, when a writer departs, both waiting readers and waiting writers have equal priority.

The sleeping barber problem:

- **Problem description**



When no customer, barber sleeps.
If no waiting chair, customer leaves.

- **Solution:**

CPU scheduling

- **Resources: the things operated on by processes. Resources ranges from CPU time to disk space, to I/O channel time.**

- **Resource fall into two classes:**

  - **Preemptive:**
    Can take resource away. Use it for something else, then give it back later.

    Examples: processor or I/O channel.

  - **Non-preemptive:**
    Once given, it cannot be reused until process gives it back.
    Examples: file space and terminal.

  - **Anything is preemptive if it can be saved and restored.**

- **O.S. makes two related kinds of decisions about resources:**
  - **Allocation: Who gets what.**
    Given a set of requests for resources, which process should be given which resources in order to make most efficient use of the resources.
  - **Scheduling: How long can they keep it.**
    When more resources are requested than can be granted immediately, in which order should hay be served?
    Examples: processor scheduling and memory scheduling (virtual memory).
- **Resource # 1: the processor.**
- **Processes may be in any one of three general scheduling states:**
  Running
  Ready
  Blocked

- **Criteria for a good scheduling algorithm:**

  1. **Fairness: every process gets its fair share of CPU.**
  2. **Efficiency (utilization): keep CPU busy.**
  3. **Response time: minimize response time for interactive users.**
  4. **Throughput: maximize jobs per hour.**
  5. **Minimize overhead (context swaps).**

- **Context switch: changing process. Include save and load registers and memory maps and update misc tables and lists.**

- **Clock interrupt: Clock interrupt occurs at fixed time interval (for 60 Hertz AC frequency, 60 times per second) and O.S. scheduler can run. Every interrupt is called a clock tick (a basic time unit in computer systems).**

CPU scheduling algorithms

- **First in first out (FIFO or FCFS)**
  - **Run until finished, usually "finished" means "blocked." Process goes to the back of run queue when ready.**
  - **Problem: one process can dominate the CPU.**
  - **Solution: limit the maximum time that a process can run without a context switch. The time is called quantum or time slice.**

- **Round robin scheduling**
  - **Maintain a list of runnable processes.**
  - **Run process for one quantum then move to the back of queue. Each process gets equal share of the CPU.**
  - **If process blocks (say for I/O or semaphore) than remove it from queue and start running next process in queue.**
  - **If process becomes runnable, add it to end of queue.**
  - **Length of quantum:**

    Short: too much overhead
    Long: poor response time
    In general: about 100 ms (or about 10K -100K instructions)

- **Priority scheduling**
  - **Assign each process a priority.**
  - **Run the runnable process with the highest priority.**
  - **How to assign priority:**
    I/O bound jobs have higher priority
    CPU bound jobs have lower priority
    If a job uses $\frac{1}{f}$ of the quantum, then Priority $= f$.
  - **Unix command "nice."**
  - **Problem: high priority job may dominate CPU.**
  - **Solution: decrease priority of running process at each clock tick (dynamic priority).**

- **Priority classes**
  - **Combine round robin and priority.**
  - **Group processes into priority classes.**
  - **Use priority scheduling among the classes.**
  - **Use round robin within each class.**
  - **Classes may have different quantum.**
  - **Adaptively change quantum: exponential queue**
    - **Give newly runnable process a high priority and a very short quantum.**
    - **If process uses up the quantum without blocking then decrease priority by 1 and double quantum for next time.**

– **Example:**
Two processes P_1 and P_2.
P_1: doing 1 ms computation followed by 10 ms I/O.
P_2: doing all computation.
Initial quantum $= 100$ ms.

1. **P_1: priority 100 uses 1 ms CPU, blocked for 10 ms and then becomes ready again.**
   P_2: priority 100 uses 100 ms CPU, switched out.

2. **P_1: priority 100 uses 1 ms CPU, blocked for 10 ms and then becomes ready again.**
   P_2: priority 99 uses 200 ms CPU, switched out.
   ⋮

- **Round robin may produce bad results:**

  Example: Ten processes, each requires 100 quantum.

  In round robin: each takes about $1000(10 \times 100)$ **quantum to finish.**
  In FIFO, they would require average 500 quantum to finish.

- **How can we minimize the average response time (or turnaround time)?**

Review

Classical IPC problems (Cont.)

- **The readers and writers problem**
  - **Solution one: Readers have priority**
  - **Solution two: Writers have priority**
- **The sleeping barber problem**

CPU scheduling

- **Concepts:**
  - **Resources**
  - **Preemptive resources**
  - **Non-preemptive resources**
- **Two things O.S. does on resources:**
  - **Allocation**
  - **Scheduling**

- **Shortest job first**
  - **Suitable to batch system. Non-preemptive policy.**
  - **Must know the runtime or estimate runtime of each process.**
  - **All jobs are available at system start-up time.**
  - **Schedule the jobs according to their runtimes.**
  - **Optimal with respect to average turnaround time.**

– **Example of four jobs.**

Four jobs	A	B	C	D
Runtimes (min)	8	4	4	4

1. **Run in order A B C D:**

Jobs	Turnaround times
A:	8
B:	$8 + 4 = 12$
C:	$8 + 4 + 4 = 16$
D:	$8 + 4 + 4 + 4 = 20$

Average Turnaround Time
ATT $= \frac{8+12+16+20}{4} = 14$ **(min)**

2. **Run in the order B C D A:**

Jobs	Turnaround times
B:	4
C:	$4 + 4 = 8$
D:	$4 + 4 + 4 = 12$
A:	$4 + 4 + 4 + 8 = 20$

Average Turnaround Time
ATT $= \frac{4+8+12+20}{4} = 11$ **(min)**

- **General proof.**
  1. **For Four jobs.**

     Suppose the runtimes for jobs A, B, C and D are a, b, c and d respectively. Run in the order A B C D.

     Average Turnaround Time:

$$\begin{aligned}\textbf{ATT} &= \frac{1}{4}[a + (a+b) + (a+b+c) + (a+b+c+d)] \\ &= \frac{1}{4}(4a + 3b + 2c + d)\end{aligned}$$

a **has the largest coefficient** $\Rightarrow$ **must be the smallest**

2. For n jobs.

n **jobs**	J_1	J_2	...	J_n
Runtimes	T_1	T_2	...	T_n

Run in order $J_1, J_2, \ldots, J_n$.

Jobs	**Turnaround times**
J_1	T_1
J_2	$T_1 + T_2$
J_3	$T_1 + T_2 + T_3$
$\vdots$	
J_i	$T_1 + T_2 + \cdots + T_i$
$\vdots$	
J_n	$T_1 + T_2 + \cdots + T_n$

Average Turnaround Time:

$$\begin{aligned}\textbf{ATT} &= \frac{1}{n}[nT_1 + (n-1)T_2 + (n-2)T_3 + \cdots + \\ &\quad (n-i+1)T_i + \cdots + 2 \cdot T_{n-1} + T_n]\end{aligned}$$

We want to prove:
if $T_1 \leq T_2 \leq \cdots \leq T_n$, then ATT is optimal (smallest).

Prove it by contradiction.

Suppose for some $1 \leq i < j \leq n$, we have

$$T_i > T_j.$$

Note that we have $(n-i+1)T_i$ and $(n-j+1)T_j$ in ATT, but

$$n - i + 1 > n - j + 1.$$

We can always reduce the ATT by exchanging the running order of J_i and J_j. That is,

$$(n-i+1)T_j + (n-j+1)T_i < (n-i+1)T_i + (n-j+1)T_j$$

Thus, shortest job first is optimal.

– **If jobs are not available at the beginning, the shortest job first may not be optimal.**

A counterexample.

Five jobs:	**A**	**B**	**C**	**D**	**E**
Runtimes:	**2**	**4**	**1**	**1**	**1**
Arrive times:	**0**	**0**	**3**	**3**	**3**

1. **Run in order A B C D E (shortest job first).**

Jobs	**Turnaround times**
A	2
B	$2+4=6$
C	$6-3+1=4$
D	$4+1=5$
E	$5+1=6$

ATT $= \frac{2+6+4+5+6}{5} = \frac{23}{5}$

2. Run in order B C D E A.

Jobs	Turnaround times
B	4
C	$4-3+1=2$
D	$2+1=3$
E	$3+1=4$
A	$4+3+2=9$

ATT $= \frac{4+2+3+4+9}{5} = \frac{22}{5}$

– **How to use shortest job first in an interactive system**

  * **Consider each command as a job**

  * **Estimate the runtime for a command (job) based on past runtime.**

    T_0: **estimated runtime per command for some terminal**
    T_i: **runtime for the** i**th command** $(i \geq 1)$
    S_i: **predicted runtime for the** i**th** $(i \geq 1)$ **command**

    Recurrence:

$$
\begin{aligned}
S_1 &= T_0 \\
S_{n+1} &= aT_n + (1-a)S_n, \quad 0 \leq a \leq 1, n > 1
\end{aligned}
$$

    Closed form:

$$
\begin{aligned}
S_{n+1} = \; & aT_n + (1-a)aT_{n-1} + \cdots + \\
& +(1-a)^i aT_{n-i} + \cdots + \\
& +(1-a)^{n-1} aT_1 + (1-a)^n T_0
\end{aligned}
$$

Let $a = \frac{1}{2}$**,**

$$
\begin{aligned}
S_{n+1} &= \frac{1}{2}T_n + \frac{1}{2^2}T_{n-1} + \cdots + \\
&\quad + \frac{1}{2^{i+1}}T_{n-i} + \cdots + \frac{1}{2^n}T_1 + \frac{1}{2^n}T_0
\end{aligned}
$$

- **Aging algorithm:**

  Estimate the next value in a series by taking the weighted average of the current measured value and previous estimate.

- **Possibility of starvation of longer jobs.**

- **Guaranteed scheduling (fair share scheduling)**

  n **users logged in, each user receives about** $\frac{1}{n}$ **of the CPU time.**

  - **Keep track of:**

    How long each user logged in

    How much time a user used

  - **Compute:**

    CPU time entitled for a user $= \frac{\text{logged in time}}{n}$

    Ratio $= \frac{\text{used time}}{\text{entitled time}}$

  - **Choose the lowest ratio process to run until its ratio has moved above its closest competitor.**

- **Two-level scheduling**
  - **Lower-level: scheduling among the processes in memory**
  - **Higher-level: scheduling between disk and memory**

Criteria for higher-level scheduling:

1. **Time the process stays in memory**
2. **CPU time of the process**
3. **Size of process**
4. **Priority of the process**

- **Summary for scheduling algorithms:**
  - **In principle, scheduling algorithm can be arbitrary, since the system should produce the same results in any event (get the job done).**
  - **However, the algorithms have strong effects on the system's overhead, efficiency and response time.**
  - **The best schemes are adaptive. To do absolutely best, we have to be able to predict the future.**
  - **Best scheduling algorithms tend to give highest priority to the process that needs the least!**