Note Read chapter 12 of textbook!

I Task Scheduling

- In a multitasking system, the main objective is to have the CPU utilized at all times (in practical systems 40% ÷ 90%)
- At each opportunity, OS asks: If N tasks are Ready, which one should I run?
- If we can show that the scheduler always can achieve deadlines, the system is "deterministically schedulable"!
- In real-time design, absolutely essential are:
  - repeatability
  - predictability
- Scheduling types:
  1) - non-preemptive ≡ a new task that is Ready is scheduled only after the currenty Running one switches to Waiting state (initiated by an I/O request) or after it terminates.
  2) - preemptive ≡ otherwise.

- Goals
  1) CPU utilization $U_{CPU} = 1 - \frac{idle}{period}$.
  2) Throughput: work units completed per unit of time. (# of processes)
  3) Turnaround time: interval of time from submission of a task and its completion (waiting + execution + I/O time)
  4) Waiting time: time spent waiting in the queues. Recall that when a task enters the system initially, it is put into an "entry queue". Then, it is placed in "ready queue", when conditions are met.

5) Response Time: the time from submission to first response.

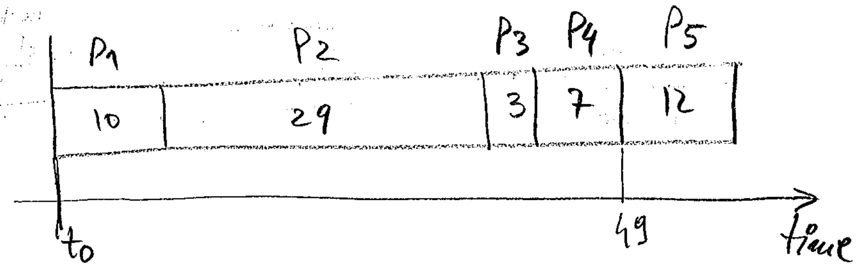
Scheduling Algorithms in Time-Shared Systems

1 First-come First-Served

- when process enters the system, it is added to a (FIFO) queue.
- when the CPU becomes free, it is allocated to the process at the head of the queue.
- it's non-preemptive

Example:

process	(Burst) Time
P1	10
P2	29
P3	3
P4	7
P5	12



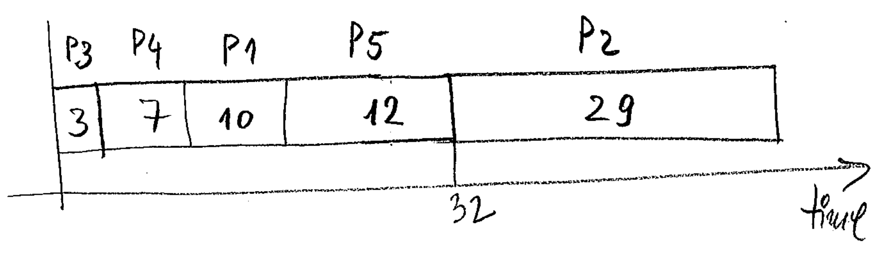
Average wait time:

P1	0
P2	10
P3	32
P4	42
P5	49

$\boxed{28}$

2 Shortest Job first

- each task has associated with it an estimate of how much time will need to complete its execution once given the CPU
- this algorithm can be
  - preemptive (Running task may be interrupted by shorter task)
  - non-preemptive



Average Wait Time

P3	0
P4	3
P1	10
P5	30
P2	32

$\boxed{13}$

3 Round Robin

- designed especially for time shared systems.
- similar to first-come first-served, with preemption added to switch between processes.
- time quantum (or slice) ≡ time "unit" for which CPU is allocated to tasks or processes.
- Ready queue is treated as a circular queue. Scheduler walks the queue, allocating CPU to each task for one time slice!
- If process - completes in less than a slice time => it releases CPU
  - does not complete within a slice time => it's interrupted and placed at the end of queue!

P1	P2	P3	P4	P5	P2	P5	P2
10	10	3	7	10	10	2	9

time: t_0, 10, 20, 23, 30, 40, 50, 52, 61

Assumption: time slice = 10 time units

	Average waiting Time
P1	0
P2	32
P3	20
P4	23
P5	40
	$\boxed{23}$

32 = 10 + 20 + 2

$\boxed{\text{Note}}$: if the evaluation criterion is "average waiting time", then shortest-job-first should be selected as the scheduling algorithm in the example above!

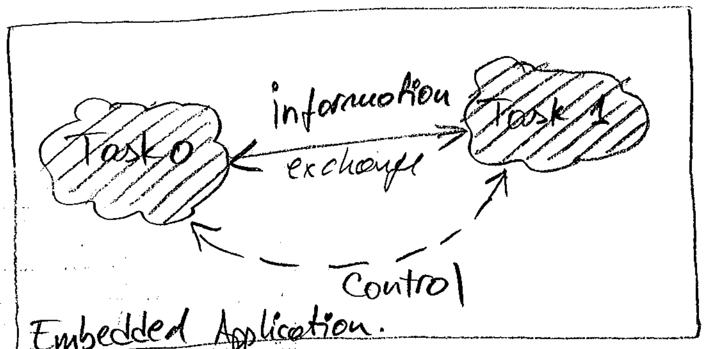
II Tasks and Threads Communication

- Resource sharing and intertask communication and synchronization must take place in a robust, safe, and reliable manner!

- The $\boxed{\text{model}}$ for interprocess communication and synchronization:

  moved via shared variables or messages!

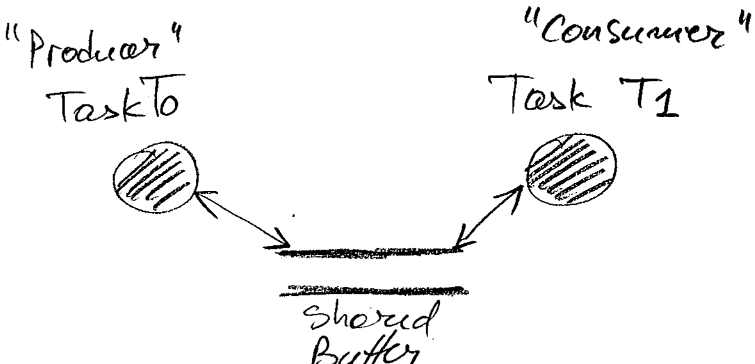
  1. The information ≡ data or signals being moved (variables or pointer variables w/ memory address)
  2. Place (or places) from which (data) information is moved to/from
  3. Control and synchronization of actions and the movement of information (techniques based on flags or status bits)



1.a) Shared Variables

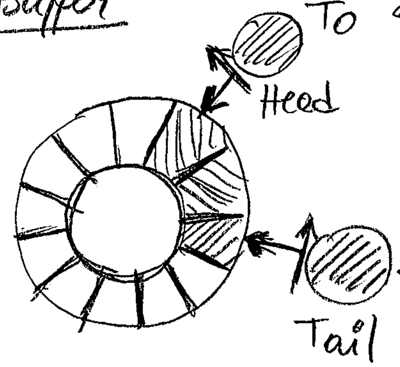
→ Global Variables
- advantage: do not need to be copied on the stack during a context switch!

→ Shared Buffer = an exchange technique in which two processes share a common set of memory locations.



<u>issue</u>: if tasks are running at different speeds, there is the potential for <u>overrun</u> or <u>underrun</u>!
=> need appropriate <u>buffer sizing</u>!

→ <u>Ring Buffer</u>



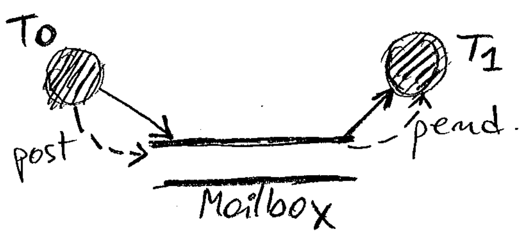
T_0 ← Adds data to the Buffer

T_1 ← Removes data from Buffer.

<u>issue</u>: underflow, overflow.

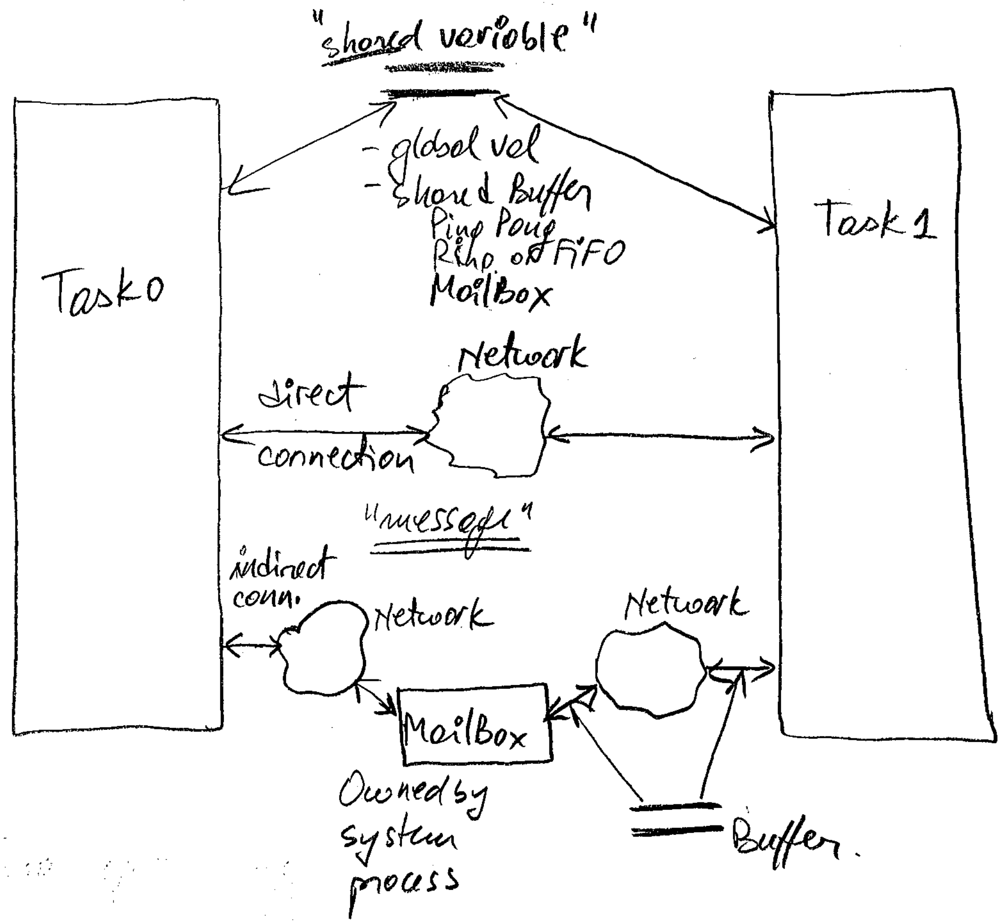
→ <u>Mailbox</u>
(similar to queue)
- supports two operations
  1) <u>write</u> (or <u>post</u>): during which a flag is raised to indicate data is available.
  2) <u>read</u> (or <u>pend</u>)

- Data flow and control diagram:



1.b <u>Messages</u>
- Utilized especially in <u>distributed</u> applications.
- supports 2 operations:
  1) send
  2) receive
- messages can have <u>fixed</u> or <u>variable</u> sizes!
- messages can be moved from one place to another <u>directly</u> or <u>indirectly</u>.

Summary of intertask Communication:



III Task Cooperation, Synchronization, and Sharing

- Many times, in a multitasking system, processes need to cooperate/synchronize as they execute the application!

- A "critical section" is a resource that several tasks may share such as an I/O port or segment of memory in which they are reading and writing common variables.

- We must ensure mutually exclusive access to these resources!

- Any solution to the critical section problem must satisfy the requirements:
  1) ensure mutual exclusion.
  2) prevent deadlock
  3) ensure progress
  4) ensure bounded waiting

Semaphores

- in simplest form is a Boolean variable or an integer S that can be accessed through only 2 atomic operations:

  1) wait(S)
  2) signal(S)

atomic — guaranteed to terminate and indivisible

```
wait (S)
{
    while (S);
    S = TRUE;
}
S initialized to FALSE
```

```
signal (S)
{
    S = FALSE;
}
```

```
Task T0
{
    ...
    wait(S)
    critical section
    signal(S)
    ...
}
```

```
Task T1
{
    ...
    wait(S)
    critical section
    signal(S)
    ...
}
```

- Semaphore S can be used to protect a critical resource!