2.2.6 Monitors:

What happens if the two Waits in the producer code (producer-consumer solution #1) were reversed in order ?

Assume buffer is completely full.

Deadlock !

It is very easy to make errors using semaphores.

Problems that result are very difficult to debug (races and deadlocks).

Monitors make it easier to write correct programs.

Monitor: collection of procedures, variables and data structures packaged and restricted like a class in C++, e.g.,

```
monitor example
   {...
   procedure producer(x);
   procedure consumer(x);
   };
```

Property: Only one process can be active in a monitor at any instant.

Operating Systems Processes CMSC 421

Compiler handles calls to monitor code differently

First couple lines of each procedure check to see if another process is active in the monitor.

If so, current process is suspended.

Users turn critical section into **monitor** procedures.

Compiler takes care of implementing mutual exclusion.

Compiler is less likely to get it wrong.

What about synchronization ? Processes that block within the monitor will prevent others from entering.

Wait() and Signal() operations on **condition variables** allow processes to block. Also allows waiting processes to enter monitor.

Special rules prevent more than one process from being active in a monitor.

Operating Systems Processes CMSC 421

2.2.7 Message Passing:

Problems with monitors:

Since monitors are a programming language construct, compiler must be aware of them. C and pascal do not have them.

Monitors depend on shared memory - will not work in a distributed environment with multiple CPUs connected via LAN.

Message passing provides a solution to these problems.

Two primitives:

send: send a message to a given destination, e.g.

send(destination, &message);

receive: receive a message from a given source (or ANY), e.g.

receive(source, &message);

receive could block if no message is available OR

receive could return immediately with an error code.

Like semaphores and unlike monitors, send and receive are system calls.

Operating Systems Processes CMSC 421

Problems:

- Messages could be lost. Acknowledgements and sequence numbers are used to guard against this problem.
- Process naming and authentication problem.
- Performance issues when sender and receiver are on same machine.

Look over simple implementation of producer-consumer solution using message passing.

2.3.2 Readers and Writers

Suppose we have shared data that a group of processes can both read and write.

Readers can share access with each other, but writers cannot share access with either readers or other writers.

One solution: a potential writer must wait for all readers to finish.

Problem: writers can starve, if a succession of readers arrive.

Another solution: if a writer is waiting, then no other reader can gain access to the shared object.

Problem: readers can starve.

Starving writers

Suppose we have two semaphores, **mutex** and **wrt**, and a variable **readcnt**.

Initially, mutex = wrt = 1, readcnt = 0

Readers

```
while (1)
   {
   Wait(&mutex);
   readcnt = readcnt + 1;
   if (readcnt == 1)
      Wait(&wrt);
   Signal(&mutex);
   read shared data;
   Wait(&mutex)
   readcnt = readcnt - 1;
   if (readcnt == 0)
      Signal(&wrt);
   Signal(&mutex);
   }
```

Writers

```
while (1)
   {
   Wait(&wrt);
   write to shared data;
   Signal(&wrt);
   }
```


Analysis:

Multiple writers use wrt for mutual exclusion.

Suppose there is no active writer.

		wrt	readcnt	
Initially		1	0	
reader 1:	readcnt++	1	1	
reader 1:	Wait(&wrt)	0	1	(now any writer must wait)
reader 2:	readcnt++	0	2	

Note:

First reader locks out the writers.

Second reader does not do a Wait(&wrt).

Any number of readers can now keep the writers locked out, until the last reader executes Signal(&wrt).

		wrt	readcnt	
eventually:		0	1	
reader k:	readcnt--	0	0	
reader k:	**Signal**(&wrt)	1	0	(writers can write again)

If there is an active writer and some readers arrive, then

		wrt	readcnt
		0	0
reader 1:	readcnt++	0	1
reader 1:	**Wait**(&wrt)	-1	1

reader 1 waits before executing Signal(&mutex).

This causes subsequent readers to wait on mutex.

Look over Dining Philosophers Problem and Sleeping Barber Problem.

2.4 Process Scheduling

The **scheduler** is a part of the kernel that must decide which one of several runnable processes is to be run next.

Possible scheduling criteria:

- **Waiting time**: the sum of the periods spent waiting in the ready queue. Fairness ensures each process gets its fair share of the CPU.
- **Efficiency**: CPU utilization should be near 100%.
- **Response time**: time from the submission of a request until the first response is produced.
- **Turnaround time**: interval from the time of submission to the time of completion.
- **Throughput**: number of processes completed per time unit.

Some of these criteria are contradictory, e.g. Response time (interactive users) and Turnaround time(batch users).

Objective: Maximize CPU utilization and throughput. Minimize turnaround time, waiting time and response time.

Scheduling Decisions:

Scheduling decisions made under four circumstances:

1) Process switches from running to waiting (I/O or Wait()).
2) Process switches from running to ready (Timer).
3) Process switches from waiting to ready (I/O completion or Signal()).
4) Process terminates.

For cases 1 and 4, scheduler MUST select a new process to run.
For cases 2 and 3, there is a choice.

If scheduling takes place only for cases 1 and 4, then the scheduling scheme is nonpreemptive.
If scheduling takes place for cases 2 and 3, then its preemptive.

Nonpreemptive scheduling: A strategy that allows processes to keep CPU until they terminate or block for I/O. Simple but inappropriate for multitasking.

Preemptive scheduling: A strategy of allowing processes that are logically runnable to be temporarily suspended.

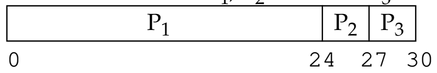
Scheduling Algorithms

First-Come, First-Served Scheduling (FCFS)

- Nonpreemptive algorithm.
- Simple implementation using FIFO queue.
- Problem: Average waiting time not optimal, e.g.
  Suppose processes P1, P2 and P3 arrive simultaneously.

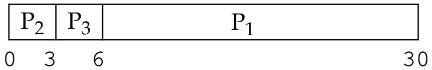
Process	CPU-burst time (milliseconds)
P_1	24
P_2	3
P_3	3

If arrive in order P_1, P_2 and then P_3:



Average wait time = 17 milliseconds

If arrive in order P_2, P_3 and then P_1:



Average wait time = 3 milliseconds

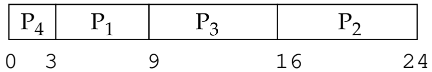
Gantt chart analysis.

Not appropriate for time-sharing systems.

2.4.4 Shortest Job First (SJF)

- When CPU available, process with smallest next CPU burst assigned.
- Appropriate for batch jobs where run times are known in advance.

Process	CPU-burst time (milliseconds)
P_1	6
P_2	8
P_3	7
P_4	3



Average wait time = 7 milliseconds

The method is optimal in giving the minimum average waiting time for a given set of processes.

The SJF algorithm may be either preemptive or nonpreemptive.

Preemptive SJF algorithm

Good for time-sharing systems since it produces the minimum average response time.

Problem: Determining which of several runnable processes is shortest.

Consider repeated operation sequence of “command - execute command” typed at multiple terminals by users.

We can derive an estimate of the run time of a new *command* by using the past behavior at each terminal. e.g.

Suppose the estimated time per command for some terminal is T_0.

Suppose its next run is measured to be T_1.

We could update the estimate using the weighted sum:

$$aT_0 + (1-a)T_1$$

where a determines how fast we *forget* old runs (memory).

In preemptive SJF, a choice arises when a new process arrives at the ready queue.

Process	Arrival Time	CPU-burst time (milliseconds)
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

Average waiting time:

$$\overset{P_1}{(10-1)} + \overset{P_2}{(1-1)} + \overset{P_3}{(17-2)} + \overset{P_4}{(5-3)})/4 = 6.5 \text{ msecs}$$

Nonpreemptive version is 7.75 milliseconds

Process P_1 is preempted at time 1 because process P_2 can finish before P_1 can finish.

2.4.2 Priority Scheduling

- Each process is assigned a priority and the runnable process with the highest priority is allowed to run.

All processes arrive at time 0.

Process	Priority	CPU-burst time (milliseconds)
P_1	3	10
P_2	1	1
P_3	3	2
P_4	4	1
P_5	2	5

P_2	P_5	P_1	P_3	P_4

0 1 6 16 18 19

Average wait time = 8.2 milliseconds

SJF is a special case of priority scheduling in which the priority is the inverse of the (predicted) next CPU burst.

Priority scheduling may be either preemptive or nonpreemptive.

Problem: Indefinite blocking or **starvation**.

In order to prevent starvation, the scheduler may decrease the priority of the currently running process at each clock tick.

Other characteristics:

- Priorities can be assigned statically or dynamically.
- OS can assign priorities dynamically based on the class of the job.
  - Highly I/O bound processes should be given high priority when they become ready to run again.
  - I/O bound processes typically do not use up their time slice. Why ?
  - Simple priority of 1/f, where f is fraction of the last quantum used, could be used by the OS where larger values mean higher priority.

2.4.1 Round Robin Scheduling

- Similar to FCFS scheduling but preemption is added.
- Ready queue is treated as a circular queue.
- Each process assigned a **time slice**.
- Timer interrupt preempts running process and OS puts process on the back of a FIFO queue.
- Only interesting issue is length of the time slice.
  - If a context switch takes 5 msec and the time slice is 20 msec, then 20% of CPU time wasted on performing context switch.
  - Making it too long (e.g. 500 msec) on an interactive system can lead to user frustration due to slow response time.

Time slice of 4.

Process	CPU-burst time (milliseconds)
P_1	24
P_2	3
P_3	3

All processes arrive time 0.

P_1	P_2	P_3	P_1	P_1	P_1	P_3	P_4
0–4	4–7	7–10	10–14	14–18	18–22	22–26	26–30

Average wait time =5.66 milliseconds

2.4.3 Multilevel Queue Scheduling

- Appropriate when process can be easily grouped into priority classes, e.g. foreground vs background processes.
- Each queue has its own scheduling algorithm.
- There must be scheduling between queues.

One possible implementation is to run all processes in highest priority queue until it becomes empty, then run next lower priority queue processes.

- Leads to starvation of lower priority processes if priorities are not adjusted occasionally.
- **Multilevel feedback queue scheduling**: Processes are moved between queues (priority adjusted) based on their activities, e.g. I/O activity, terminal (user) activity or CPU (compute bound) activity.

2.4.5 Guaranteed Scheduling

- Make commitments to the user about performance, e.g.,

  If n users on system, users are promised 1/n of the CPU power.

  In order to accomplish this, OS must keep track of how much CPU time each process has had since its creation.

  The amount of time each process is entitled to is simply the time since its creation divided by n.

  Ratios of time actually used to time entitled that are less than 1 indicate that the processes priority should be increased. The opposite is true for ratios greater than 1.

2.4.6 Lottery Scheduling

- Gives similar results to guaranteed scheduling but is simpler to implement.

  Give process lottery tickets for various system resources, such as CPU time.

  Scheduling decisions are made by selecting a random lottery ticket.

  Important processes can be given more tickets.
- More straightforward than priority scheduling. If a process holds 20% of the tickets, in the long run, it gets 20% of the resource. How do we translate a priority of 40 into resource time usage ?
- Advantages:

  It is highly responsive to new processes entering the system.

  Cooperating processes can exchange tickets, e.g. client process blocking on a resource could give all its tickets to the server managing the resource.

2.4.7 Real-Time Scheduling

- CPU must react to (typically, external device) requests within a fixed amount of time, e.g. CD player generates bits that must be converted into sound.
- Two types of systems:
  Hard real time: Absolute deadlines.
  Soft real time: Missing an occasional deadline is tolerable.
- In either case, real time behavior is achieved by dividing the program into a number of processes, each of whose behavior is predictable and known in advance.
- When an external event occurs, the scheduler's job is to determine an execution sequence that succeeds in meeting all deadlines.
- Events may arrive
  periodically: at regular intervals
  aperiodically: unpredictably

- A system that must respond to multiple periodic event streams, m, requires

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le 1$$

where event i occurs with period P_i and requires C_i seconds of CPU time to handle, otherwise the real-time system is not schedulable.

- Real-time scheduling algorithms can be
  dynamic: scheduling decisions made at run time.
  static: scheduling decisions before the system is running.

- Examples of dynamic real-time scheduling algorithms:
  Rate monotonic algorithm, **earliest deadline first** and **least laxity** algorithm.

In general, converting general purpose OSs into real-time systems by using one of these algorithms is not useful because of context-switch overhead.