Operating systems 1, Lecture 11

Sanda-Maria AVRAM, Ph. D.

Concurrent processes

1

A program is a sequence of instructions describing the operations to be performed during a run.

A process has a *dynamic* character; is a set of program activities that work according to the context.

The execution control of concurrent processes

2

Parallel processes are those processes that can be executed in parallel; these processes do not interrelate during execution, and do not work together.

Concurrent processes are those processes that can interrelate during parallel execution, for example by sharing resources.

Concurrence between processes

3

```
int n;
int fd = open("seats.db", "ORDWR");
read(fd, &n, sizeof(int));
lseek(fd, 0, SEEK_SET);
n--;
write(fd, &n, sizeof(int));
close(fd);
```

Critical section. Critical resource

4

The process A	n = (in A)	The process B	n = (in B)
read(fd, &n, sizeof(int));	10		
	10	read(fd, &n, sizeof(int));	10
n--	9		10
lseek(fd, 0, SEEK_SET)	9		10
write(fd, &n, sizeof(int));	9		10
		n--	9
		lseek(fd, 0, SEEK_SET)	9
		write(fd, &n, sizeof(int));	9

Critical section. Critical resource. Mutual exclusion

5

The process A	n = (in A)	The process B	n = (in B)
read(fd, &n, sizeof(int));	10		
n--	9		
lseek(fd, 0, SEEK_SET)	9		
write(fd, &n, sizeof(int));	9		
		read(fd, &n, sizeof(int));	9
		n--	8
		lseek(fd, 0, SEEK_SET)	8
		write(fd, &n, sizeof(int));	8

The Semaphore Concept

6

- Semaphore = (v(s), c(s))
- Operations:

P(s) – WAIT – Allocation

```
v(s) = v(s) - 1;
if(v(s) < 0) {
    STATE(A) = WAIT;
  //process A goes into standby mode
    c(s) <- A;
    <Passes control to the DISPATCHER>;
}
else{
    <Passes control over the process A>;
}
```

V(s) – SIGNAL – Deallocation

```
v(s) = v(s) + 1;
if(v(s) <= 0) {
    //extracts process B from the queue
    c(s) -> B;
    STATE(B) = READY;
    <Passes control to the DISPATCHER>;
}
else{
    <Passes control over the process A>;
}
```

Important questions

7

1. Does the semaphore order matter? YES
2. Does the order of P operations matter? YES
3. Does the order of V operations matter? NO

YES = deadlock

The problem of the producer and the consumer

8

```
semaphores full, empty, excluded;
v0(full) = 0;
v0(empty) = n;
v0(excluded) = 1;
```

Producer

```
…
< produces an article >;
P(empty);
P(excluded);
< puts the article in the buffer >;
V(excluded);
V(full);
…
```

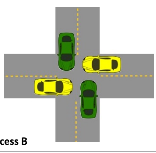
Consumer

```
…
P(full);
P(excluded);
< extract the article from the buffer >;
V(excluded);
V(empty);
< consuming the article >;
…
```

The concept of deadlock

9

- is a problem related to allocating resources between multiple concurrent processes



```
semaphore x, y;
V0(x) = 1;
V0(y) = 1;
```

Process A

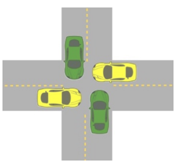
```
P(x);
<instructions of A>;
P(y);
V(y);
V(x);
```

Process B

```
P(y);
<instructions of B>;
P(x);
V(x);
V(y);
```

The concept of deadlock (cont.)

10



- Exiting deadlock
- Detecting the deadlock
- Preventing the deadlock

Exiting deadlock

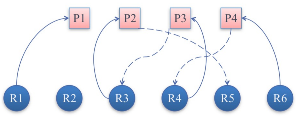
11

- is usually destructive; done by one of the following methods:
  - Reloading the operating system
  - Choosing a "victim" process
  - Creating a "resume point"

Detecting the deadlock

12

- It is done when the OS does not have a deadlock prevention mechanism
  - Resource allocation graph: **Cycle in graph** => *DEADLOCK*



Preventing the deadlock.
4 necessary conditions for a deadlock to occur

13

Deadlock can only occur in systems where **all** 4 conditions hold true:

1. Mutual exclusion - a resource cannot be used by more than one process at a time
2. Hold and wait - processes already holding resources may request new resources
3. No preemption - only a process holding a resource may release it
4. Circular wait - two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds

Preventing the deadlock.
(1) by removing the mutual exclusion condition

14

- It means that no process may have exclusive access to a resource

Algorithms that avoid mutual exclusion are called *non-blocking synchronization algorithms*.

Preventing the deadlock.
(2) by removing the "hold and wait" condition

15

All-or-none algorithms (impractical and inefficient use of resources).

- requiring processes to **request all** the resources they will need **before starting up**.
- require processes to **release all** their resources **before requesting** all the resources they will need

Preventing the deadlock.
(3) by "preemption" (lockout)

16

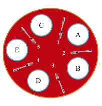
- it means that processes can have resources taken away while that resource is being used
- difficult or impossible as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent

Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.

Preventing the deadlock.
(4) by avoiding circular waits

17

- allowing processes to wait for resources, but the waiting cannot be circular:
  - assign a precedence to each resource and force processes to request resources in order of increasing precedence (e.g., **Dining Philosophers Problem**
    - According to Edsger Dijkstra and Tony Hoare:
      - Eating requires two chopsticks (more realistic than forks...)
      - A philosopher may only use the closest left and right chopsticks
    - *not practical for a larger number of resources*)



- allow holding only one resource per process; if a process requests another resource, it must first free the one it's currently holding (or hold-and-wait).

Avoiding the deadlock by resource-controlled allocation

18

For every resource request, the system sees if granting the request will mean that the system will enter an *unsafe state* that could result in *deadlock*.

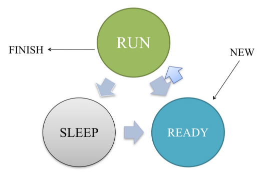
- the system must know in advance at any time the number and type of all resources in existence, available, and requested.

- **the Banker's** algorithm developed by Edsger Dijkstra
  - *the bank ensures that when customers **request** money the bank **never leaves a safe state**. If the customer's request does not cause the bank to leave a safe state, the loan will be granted, otherwise the customer must wait until some other customers deposit enough.*

The concept of multiprogramming

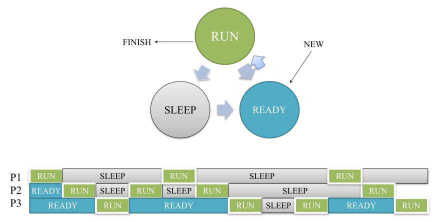
19

- Simultaneous existence in the internal memory of several processes that run concurrently



Process Planner Operation

20



The tasks of the Process Planner

21

- Keeping track of all processes in the system;
- Choosing the process to which the processor will be assigned and for how long;
- Assigning processor to a process;
- Releases the processor at the exit of the process from the **RUN** state.

The Process Planner also maintains:

22

- a data structure (called *Process Control Block* (PCB)) for each process in the system:
  - *process state*
  - *pointer to the next PCB with the same state*
  - *process number*
  - *the process counter (address of the machine instruction to be executed);*
  - *area for copy of general machine registers*
  - *the limits of the memory areas assigned to the process*
  - *the list of the open files*
- queues at I/O peripherals.

Process planning algorithms

1. FCFS (First Come First Served)
2. SJF (Shortest Job First)
3. Algorithm based on priorities
4. Algorithm based on deadlines (deadline scheduling)
5. Round-Robin (circular planning)
6. Algorithm of queues on multiple levels

Process planning algorithms
1. FCFS (First Come First Served)

- Processes are served in their chronological order.

  For example, for 3 processes that have run times: 24, 3, 3 minutes,

  - if they come in order 1, 2, 3 the average serving time will be (24 + 27 + 30) / 3 = 27 minutes
  - if they come in order 2, 3, 1, their average serving time will be: (3 + 6 + 30) / 3 = 13 minutes

- Simple but not very efficient algorithm

Process planning algorithms
2. SJF (Shortest Job First)

- Executes the process with the shortest execution time.

- Disadvantage: The execution times of the processes must be known.

Process planning algorithms
3. Algorithm based on priorities

1. The process receives a priority when it enters the system and keeps it up to the end
2. OS computes priorities and dynamically assigns them to the running processes

\+ Most often used.

\- Low priority processes can wait indefinitely ... phenomenon called *starvation*

A combined method can be used, by which the low priority processes will increase their priority as long as they wait longer and return to the initial value after they are executed.

Process planning algorithms
4. Algorithm based on deadlines (deadline scheduling)

- in OSs designed with very strict real time requirements
- Each task is assigned a termination term
- The scheduler uses these terms to decide to which process will allocate the processor for all processes to complete in time.

Process planning algorithms
5. Round-Robin (circular planning)

- Designed for OS that work in time sharing.
- A quantum of time (between 10-100 milliseconds) is defined.
- The READY queue is treated circularly, each process being allocated to the processor for a quantum of time, after which the process passes to the end of the queue.

Process planning algorithms
6. Algorithm of queues on multiple levels

- Applies when works can be easily categorized into distinct groups:
  - Classification of tasks: system, interactive, text editing, etc.

The End