Processes-1
A Formal View

Processes
Flow Graphs
Determinancy

Definitions

- The notion of a sequential process helps to cope with structuring.
- It captures the notion of:
  - Non-determinism
  - Parallel activities

Informally, a sequential process or task is the activity resulting from the execution of a program by a sequential CPU

A process consists of:
- Program (the text segment)
- Data (the data segment)
- A Thread of Execution (i.e., runtime information such as program counter and stack)

In general: *A process is a program whose execution has started and not yet terminated!*

A simplified view….

- A process may be in any one of several states:
  - Running → the process is using the processor to execute an instruction.
  - Ready → the process is executable but other processes are executing and all CPUs are in use.
  - Blocked → the process awaits an event to occur:
    - Resource availability
    - Messages
    - Signals

Concurrent processes are individually scheduled to use the CPU.

Concurrent processing results from multiple instantiations (creation) of a set of processes

$P = \{p_1, p_2, p_3 \ldots p_n\}$

Each process p_i may execute a different program!

A System of Processes

Let $P = \{p_1 \ldots p_n\}$ a set of processes in the system.

We define the following:
- $D(p_i)$ is the domain of p_i
- $R(p_i)$ is the range of p_i

We can view process p_i as a function $p_i = D(p_i) \rightarrow R(p_i)$ that maps memory to memory.

Given two processes, p_i and p_j, we must consider the possibility of interference.

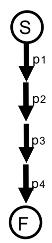
i.e., the order of execution of p_i and p_j may matter!

We can prescribe the execution order of p_i and p_j by a precedence relation "➔".

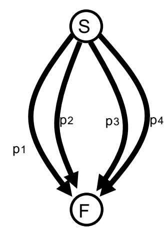
➔ = $\{(p_i, p_j)$: p_i must complete before the start of $p_j\}$

Process Flow Graph

The precedence constraints among a set of processes can be visualized by a process flow graph.



S(p_1,p_2,p_3,p_4)



P(p_1,p_2,p_3,p_4)

S and F denote the start and finish of the flow graph, respectively.

Every PFG is a directed acyclic graph (DAG) [why?]

S(p_1,p_2,…,p_n) denotes the sequential execution of processes.

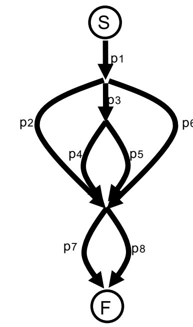
P(p_1,p_2,…,p_n) denotes the parallel execution of processes.

…more flow graph

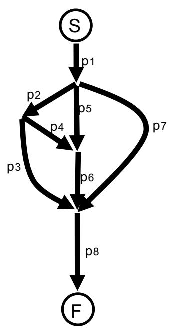
A process flow graph is properly nested if it can be described by the functions **S** and **P** and only function composition.

Example:

S(p_1,P(p_2,S(p_3,P(p_4,p_5)),p_6),P(p_7,p_8))



…general precedence..



Why is this not properly nested ?

Note that not all flow graphs are properly nested.

i.e., the precedence relation may not be expressible though **S** and **P** operations in conjunction with function composition.

Nevertheless, it is still possible to express the process precedence by using fork() and join() operations.

Determinancy

Recall the precedence relation:

➔ = {(p_i,p_j): p_i must complete before the start of p_j}

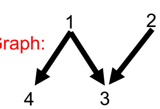
Determinancy: if all executions allowed under "➔" relation result in the same values for all memory cells, then the system is said to be determinate.

Example (non-determinate):

P_1: $C(M_1)+1 \rightarrow M_2$; $C(M_2)-1 \rightarrow M_3$ (concurrent)
P_2: $2*C(M_2) \rightarrow M_4$
P_3: $C(M_3)+C(M_4) \rightarrow M_5$
P_4: $C(M_2)+1 \rightarrow M_6$

➔ = { (1,3), (2,3),(1,4) }

Precedence Graph:



p_1 and p_2 are independent since (p_1,p_2) is not in ➔

Example…

Example (non-determinate):

Initialize: $0 \rightarrow M_1..M_6$
P_1: $C(M_1)+1 \rightarrow M_2$; $C(M_2)-1 \rightarrow M_3$ (concurrent)
P_2: $2*C(M_2) \rightarrow M_4$
P_3: $C(M_3)+C(M_4) \rightarrow M_5$
P_4: $C(M_2)+1 \rightarrow M_6$

Conflict
p_1 and p_2 conflict if:

$(D(p_1) \cap R(p_2)) \cup$
$(D(p_2) \cap R(p_1)) \cup$
$(R(p_1) \cap R(p_2)) \neq \emptyset$

	M_1	M_2	M_3	M_4	M_5	M_6
if $p_1 \rightarrow p_2$	0	1	-1	*2*	*1*	2
if $p_2 \rightarrow p_1$	0	1	-1	*0*	*-1*	

…more determinancy

In summary: A set of processes is determinate, if, given the same input, the same results are produced regardless of the relative speeds of executions of the processes or the legal overlaps in execution.

However, determinancy is not a particularly useful property as it is difficult to determine whether a large system of processes is indeed determinate.

Recall that a process can be viewed as a function:

$f_p = D(p_i) \rightarrow R(p_i)$

Supplying f_p is referred to as giving an interpretation.

→→→→ This requires the use of a stronger condition on a system of processes: *mutual non-interference*

mutual non-interference

Two processes p_i and p_j are mutually non-interfering if:

$\rightarrow^*$ means ordered directly or indirectly by $\rightarrow$

$P_i \rightarrow^* p_j$ or $P_j \rightarrow^* p_i$ or

$$\left.\begin{array}{l}(D(p_i) \cap R(p_j)) \cup \\ (D(p_j) \cap R(p_i)) \cup \\ (R(p_i) \cap R(p_j)) = \emptyset\end{array}\right\}$$

These conditions are known as Bernstein Conditions in a Database context

A system of processes is mutually non-interfering if any two processes meet the above conditions!

Important Theorems

Theorem 1:

A mutually non-interfering (MNI) system of processes is derminate (DET)

MNI → DET

However, the converse is not true: DET ¬→ MNI

Theorem 2:

Consider a system of processes in which, for each process p_i, $D(p_i)$ and $R(p_i)$ are given but the interpretation f_p is left unspecified. If the system is determinate for all interpretations, then all processes are MNI.

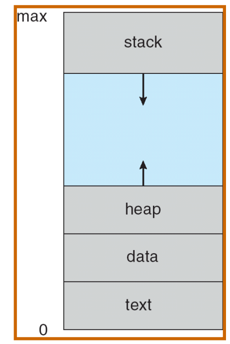
PROOF IT !!!

(I mean you in the back row…..)

Process Concept – an applied view

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

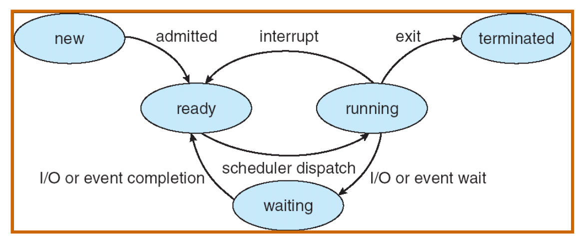
Process in Memory



Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State

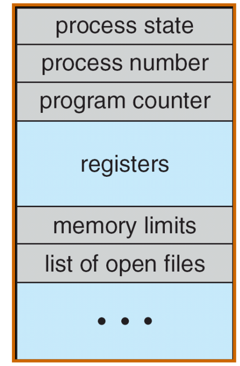


Process Control Block (PCB)

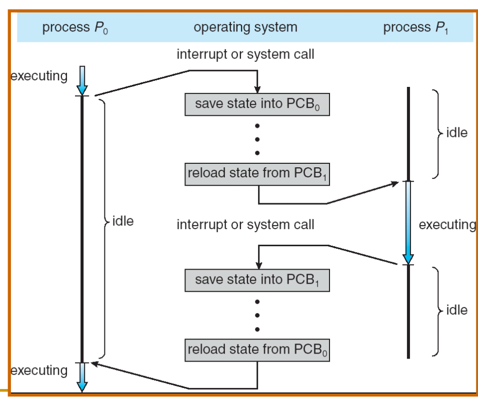
Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)



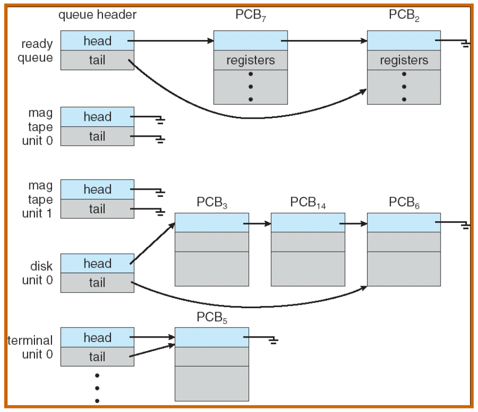
CPU Switch From Process to Process



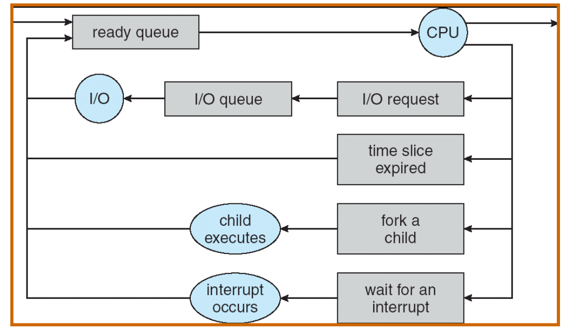
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



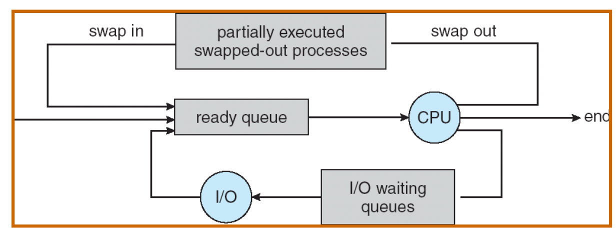
Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Addition of Medium Term Scheduling



Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

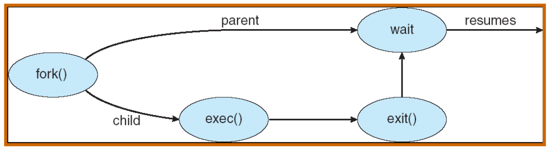
Process Creation

- Parent process create child processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

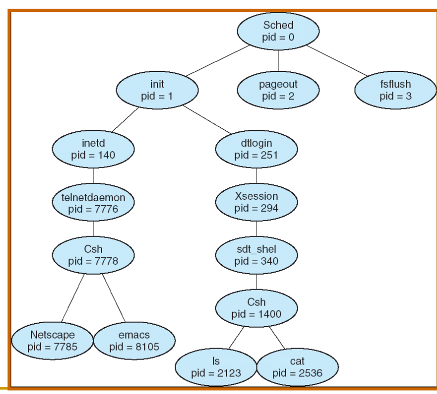
Process Creation



C Program Forking Separate Process

```
int main()
{
Pid_t  pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
   }
   else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
   }
   else { /* parent process */
        /* parent will wait for the child to
   complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
   } }
```


A tree of processes on a typical Solaris



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience