Chapter overview

- Introduction to processes, process control blocks
- Introduction to process scheduling
- Operations on processes
- Cooperating processes; threads; interprocess communication
Processes:
Review of Terminology

- **Multiprogramming**: several users share system at same time
  - batched: keep CPU busy by switching in other work when idle (e.g., waiting for I/O)
- Multitasking (timesharing): frequent switches to permit interactive use (extension of multiprogramming)
- **Multiprocessing**: several processors are used on a single system

Multiprocessing

- Multiprocessor systems: multiple CPUs, generally MIMD
  - Symmetric: identical copy of OS; communicate as necessary
    - Tightly coupled: share main memory
    - Loosely coupled: connected via communications links
  - Asymmetric: each processor has specific task
    - e.g., master/slave, channels, etc.
Terminology

• Opposite terms
  – multiprogramming and uniprogramming
  – multiprocessor and uniprocessor

• Orthogonal terms
  – multiprogramming and multiprocessor

Process

• **Process**: (Sequential) process is a program in execution. Sequential because at any time at most one instruction is in execution for a process.
• **Program**: passive entity. Static. Code.
• Process: active entity. Dynamic.
• Program and sequential process similar but not identical since one program can require multiple processes.
Sequential Process Characteristics

- Sequential
- Formed from running code plus environment
- Environment encoded in
  - Program counter
  - Process stack
  - Global data section
- Execution stream
  - Sequence of instructions performed by a process+environment

Process States

- **New**: the process is being created
- **Running**: instructions are being executed
- **Waiting**: the process is waiting for some event to occur (*such as?*). Sometimes called **blocked**.
- **Ready**: Waiting to be assigned to a processor.
- **Terminated**: Finished execution
Notes on Process States

- In uniprocessor, at most one process can be running.
- Many can be ready or waiting (or new or terminated).
- (Short term) scheduler (also called dispatcher) figures out which process is to be moved from ready to running states.
- Timer can cause process to move from running to ready states when time slice (quantum) expires.
- Process requests transfer from running to waiting by for example invoking I/O system call. Remaining transitions are OS-invoked. Wakeup occurs when request is satisfied (transfer from waiting to ready queues).
### Process Control Block (PCB)

- Information associated with each process
  - **Process state**
  - **Program counter**: next instruction to be executed
  - **CPU registers**: accumulators, index registers, stack pointers, general purpose registers, condition codes
  - **CPU scheduling information**: priorities, queue pointers, etc.
  - **Memory-management information**: base and limit registers, page/segment tables
  - **Accounting information**: resources used, account numbers, etc.
  - **I/O status information**: allocated devices, open files, etc.
  - Other information: process id, parent’s id, configuration info., etc.

### Process Control Block

- Process Control Block (PCB) also called “process descriptor” or “task control block”
- “Record” that serves as repository for descriptive information varying from process to process.
- Represents process to Operating System
- One implementation: entry in linked list where the list is associated with a particular queue (e.g., ready, running, devices, etc.)
- As process moves from queue to queue this is represented by moving the PCB from list to list
PCB Contents
(examples of possible fields)

- Process unique identifier
- current state of process
- pointer to process’ parent
- space to save needed values like program counter, CPU registers, current addressing mode (user/supervisor) when process is swapped
- CPU scheduling information (e.g., priority, scheduler data structures)
- memory management information (e.g., limit registers, page tables)
- pointers to allocated resources. I/O status information (e.g., devices, list of open files)
- accounting information (CPU time used, wall clock time used, time limits, account numbers, etc.)
- Configuration information (e.g., processor process is running on, etc.)

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
PCBs and Queues

Ready

Running

Waiting

Disk I/O

Process Scheduling

- How is process state implemented?
  - PCB moves between queues
    - State: new  Queue: job queue
    - State: ready  Queue: ready queue
    - State: waiting  Queues: device queues waiting for process termination

- How do processes move from state to state?
  - Schedulers
    - Part of the OS
    - implement a scheduling strategy (a policy)
Process Scheduling

- **Long-term scheduler** (*job scheduler*): selects processes from the pool of available processes and loads them into memory for execution
  - Which jobs should be allowed to compete actively for the resources of the system?
- **Short-term scheduler** (*CPU scheduler*): selects from among the ready processes and allocates the CPU to one of them
  - Which ready process should be assigned to the CPU?
Process Scheduling

- Short-term scheduler
  - may be executed frequently (every 100 milliseconds or so)
  - must be very fast
- Long-term scheduler
  - executes infrequently (perhaps minutes between executions)
  - can afford to take longer to make decisions
  - can take characteristics of process into account (I/O bound or CPU bound)
  - Goal: to obtain a good process mix of I/O and CPU bound processes
  - controls degree of multiprogramming
- Degree of multiprogramming: number of processes in memory
Possible Scheduling Objectives

- Fairness
- CPU efficiency
- Response time
- Predictability
- Turnaround
- Throughput
- Degrade gracefully
- Minimize overhead

Context Switch

- Context switch: required to move a process from or to “running” state
  - save state of old process
  - load saved state for new process
  - 1 to 1000 microseconds typically
  - time depends highly on degree of hardware support
- Expensive; scheduler must be designed taking cost into consideration
Process Scheduling

- **Medium-term scheduler**: Which processes should be allowed to compete for CPU (given that the other resources they need are available)
  - Swapping (swap in and out): remove processes from memory and active contention for CPU. (Later restore them to memory and permit execution to proceed.)
Operating System
Process Management Functions

- Process management provides services for
  - process creation and termination
  - process suspension and resumption
  - process synchronization
  - process communication
  - CPU scheduling

Process creation

- Parent process creates child processes; forms tree of processes
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
Process creation

- Execution options
  - Concurrent execution
  - Parent waits until children terminate
- Address space options
  - Child is duplicate of parent
  - Child has separate program loaded into it

Steps in Process Creation

- Load code and data into memory
- Create (empty) call stack
- Create (or assign) and initialize PCB
- Make process known to dispatcher
  - Dispatcher: portion of the OS that manages the running of processes. Responsible for deciding which process to run, when to start another, etc.
Steps in Process Creation
Second Approach (fork)

- Make sure current process is not running
- Update information in PCB if necessary
- Make copy of existing process. Do not copy pid, ppid, locks, pending interrupts, etc.). Do copy code, data, stack.
- Copy PCB of source into new process
- Make process known to dispatcher

UNIX fork()
Distinguishing Parent and Child

if(childpid=fork()) {
   /* this is the parent (fork returned child’s PID which is nonzero or “true” */
}
else {
   /* this is the child (fork returned 0, which is “false” */
}
Unix fork() example

#include <stdio.h>
main()
{
    int pid; char ch; int i, j;
    pid = fork();
    if(pid) ch = 'a'; else ch = 'b';
    for(i=1;i<=25;i++) {
        fputc(ch, stdout); fflush(stdout);
        for(j=1;j<100000;j++) ;
    }
    if(pid) {
        wait();
        fputc('
', stdout);
    }
}
Process Creation via fork()
Some Options

- parent and child execute concurrently (as in example)
- parent waits for all children to terminate via wait()
- child executes copy of parent’s code (as in example)
- child loads a new program and runs it via, e.g.,
  execve(path, argv, envp)
- In some other systems (VMS, e.g.) the OS creates new
  process, loads specified program into process, and starts it
  running instead of the user program doing this. Unix
  system() call simulates this.

Process Termination

- Process termination via own volition or as a result of
  system call from parent or root (e.g., exception)
  - Examples of exceptions?
  - What does parent need to know to accomplish this?
- Process may return information to parent on termination
- Deallocate process resources (physical and virtual
  memory, open files, I/O buffers, etc.). Who gets them?
- Cascading termination
- Orphan processes
- Zombie processes
Waiting Processes

• When process transfers from “running” state, information about it must be saved
• Save anything process might need to reuse (that might be damaged by another process)
  – Program counter
  – Processor status word (PSW)
  – Registers
• Must take care not to damage information in the process of saving it
• How about memory?

Waiting Processes

• Memory alternatives (three of many possible)
  – Trust the next process
  – World swap (move everything to disk)
  – Rely on memory protection to make sure that the other processes use different segments
• How expensive is the job of saving PCB and memory information?
Independent Processes

- **Independent process**: cannot affect or be affected by the other processes executing in the system
  - no shared state with other processes
  - execution is deterministic
    - depends only on input state
    - reproducible; given same input get same results
    - hence execution can be stopped and restarted without ill effects

Cooperating Processes

- **Cooperating processes**: processes can affect or be affected by other processes executing in system
  - state shared among other processes
  - result of execution cannot be predicted in advance because it depends on relative execution sequence
  - result of execution in nondeterministic. can vary with same input!
Why have Cooperating Processes?

- Information sharing (concurrent access)
- Computational speedup (parallel subtasks)
- Modularity (organizational reasons)
- Convenience (multitasking the individual)
- Supporting cooperating processes requires Operating Systems synchronization and communication mechanisms

Producer-Consumer problem

- Paradigm for cooperating processes
  - Producer process produces information that is consumed by a consumer process
- Variants
  - Unbounded buffer: places no practical limit on size of buffer
  - Bounded buffer: assumes there is a fixed buffer size
Bounded buffer/Shared memory

• Shared data
  
  \texttt{var} \ n;  
  \texttt{type} \ \texttt{item} = \ldots;  
  \texttt{var} \ \texttt{buffer: array [0..n-1] of item;}  
  \texttt{in, out: 0..n-1;}

Bounded buffer/Shared memory

• Producer process
  \texttt{repeat}
  
  \begin{verbatim}
  ---produce an item in nextp
  \texttt{while} \ in+1 \ \texttt{mod} \ n = \texttt{out} \ \texttt{do} \ \texttt{no-op;}
  \texttt{buffer}[in] := nextp;
  \texttt{in} := \texttt{in}+1 \ \texttt{mod} \ n;
  \texttt{until} \ false;
  \end{verbatim}
Bounded buffer/Shared memory

- Consumer process
  
  repeat
  
  \textbf{while} \textit{in} = \textit{out} \textbf{do} \textit{no-op};
  
  \textit{nextc} := \textit{buffer}[\textit{out}];
  
  \textit{out} := \textit{out} + 1 \mod \textit{n};
  
  ---consume the item in \textit{nextc}
  
  \textbf{until} false;

  \textit{Note that this solution only can fill up n-1 buffers}

Threads

- Traditional processes
  - operate independently of other processes
  - significant overhead in creation
  - significant overhead in switching
  - hence called “heavyweight processes”

- Wish to make it easy to share and access resources concurrently

- Wish to reduce overhead in “process” creation and in switching among processes
Threads

- Thread (also called a *lightweight process*, or *LWP*)
  - Has own
    - program counter
    - register set
    - stack space
  - Shares with peer threads
    - code section
    - data section
    - operating-system resources (open files, signals)
  - **Task**: name for the collective

Lightweight vs Heavyweight Processes

- Switching between threads is inexpensive due to the extensive sharing
  - Requires register set switch but no memory management
- Heavyweight process==task with only one thread
- LWP can be implemented at user level (**user-level threads**), at kernel level, or **both**.
- User-level threads fast because no OS involvement
  - but scheduling can be unfair because OS doesn’t know about multiple threads
  - entire process may have to wait if kernel not multi-threaded
Thread Scheduling

- States: ready, blocked, running, terminated
- Share CPU, only one thread at a time is running
- Thread executes sequentially
- Thread has own stack and PC
- Thread can create child threads
- If one thread blocked for system call, another can run
- Threads not independent because can access any address in task. No protection between threads because they are assumed to be cooperating, not hostile (as with traditional processes).
- Process synchronization mechanisms still required

Example applications

- Producer-consumer (shared buffer)
- Shared file system (block waiting for disk)
  - if threaded can continue onward acquiring work rather than having CPU idle
- Kernel operations--without threads only one task can be executing code in kernel at a time
Threads in Solaris 2

- Solaris 2 thread categories
  - User-level threads (kernel has no knowledge of these)
  - Lightweight processes (LWP)
    - One or more user-level threads associated with a LWP
    - User-level thread cannot accomplish work if not connected to LWP
    - Others either are blocked or waiting for a LWP
  - Kernel-level threads
    - Exactly one kernel-level thread associated with each LWP
    - Other kernel-level threads as well for other kernel functions
    - On request, kernel-level thread can be pinned to a specific processor (only that thread runs on processor and the processor is allocated to that thread)

- Task (Solaris 2 process): consists of at least one LWP and associated threads
- Tasks, user-level threads, LWP manipulated by the thread library
- Kernel-level threads scheduled by kernel’s scheduler
- CPU free to run something else when kernel-level thread blocks
Solaris 2 Threads

- Kernel thread: small data structure and stack. Switching is fast (no memory access information needs to change)
- LWP: PCB, register data, accounting information, memory information. Switching requires a fair amount of work and is slow
- User-level thread: stack and PC, no kernel resources. Switching among them is fast since kernel not involved. May be thousands of user-level threads but kernel only sees the LWP supporting them.
Interprocess Communication (IPC)

- Communication between two processes without resorting to shared variables
- Operations
  - Send(message)
  - Receive(message)
- Implementation issues include how links are established, whether more than two can participate, capacity of links, size limits on messages, link direction unidirectional or bidirectional

Why use messages?

- many applications fit sequential flow of information model naturally
- keeps processes totally separate except for messages
  - less error prone implementation
    - no invisible side effects
    - processes can’t mess with each others’ memory (also added security)
    - permits separation of implementation and enforcement of well-defined interfaces
  - separation especially appropriate when processes cannot “trust” each other (e.g., OS and user process)
  - permits distribution of processes, even across different kinds of processors on a network
IPC can be direct or indirect

- Direct: processes explicitly name each other
  - Send(P, message)
  - Receive(Q, message)
- Indirect: communication through intermediary of mailbox

Direct Communication
Producer/Consumer example

- Producer
  ```
  while(true) {
    produce data in nextp
    send(consumer, nextp);
  }
  ```

- Consumer
  ```
  while(true) {
    receive(producer, nextc);
    consume data in nextc
  }
  ```
Indirect Communication

- Messages are sent to and received from mailboxes (ports)
  - `send(A, message)`: deposit a message into mailbox A
  - `receive(A, message)`: extract a message from mailbox A
- Each mailbox has a unique id
- Two processes can communicate only if they have a shared mailbox
- A mailbox may be owned by a process or by the system

IPC Buffering

- Queue of messages attached to a link
  - Zero capacity: queue is of maximum length 0. Sender must wait for receiver (*rendezvous*)
  - Bounded capacity: finite length of $n$ messages. Sender must wait if link full
  - Unbounded capacity: infinite length. Sender never waits
- Variants: sender never waits but message lost if receiver doesn’t process it before another is sent.
- Sender delays until receives a reply (synchronous)
**IPC exception conditions**

- Sender or receiver terminates before message is processed
- Message lost. Options include
  - OS detects and resends message
  - Sender detects and resends
  - OS detects, notifies sender; sender takes appropriate action
- Scrambled messages

**Remote Procedure Calls (RPC)**

- High-level concept for process communication
- Programmer’s view is the same as for regular procedure calls
- Each RPC is implemented as a pair of synchronous send and receive statements
  - first pair transmits (and acknowledges) input parameters
  - second pair acquires (and acknowledges) corresponding results
- Another view of same process: remote procedure in implementation
  - begins with a receive to acquire actual parameters
  - ends with a send to provide results to caller
- Sun RPC encapsulates these in an event-driven structure
  - Remote procedures are implemented as set of handlers that are executed as called